# HIGH-PERFORMANCE MULTI-TRANSPORT MPI DESIGN FOR ULTRA-SCALE 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

Matthew J. Koop, B. C. S.

* * * * *

The Ohio State University

2009

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. F. Qin

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. The scalability and the performance of the MPI library if very important for the end application performance.

InfiniBand is a cluster interconnect which is based on open-standards and is gaining rapid acceptance. This dissertation explores the different transports provided by InfiniBand to determine the scalabilty and performance aspects of each. Further, new MPI designs have been proposed and implemented for transports that have never been used for MPI in the past. These designs have significantly decreased the resource consumption, increased the performance and increased the reliability of ultra-scale InfiniBand clusters. A framework to simultaneously use multiple transports of InfiniBand and dynamically change transfer protcols has been designed and evaluated. Evaluations show that memory can be reduced from over 1 GB per MPI process to 40 MB per MPI process. In addition, performance using this design has been improved by up to 30% over earlier designs. Investigations into providing reliability have shown that the MPI library can be designed to withstand

many network faults and also how to design reliability in software to provide higher message rates than in hardware. 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 in memory of my father, Dr. Allen H. Koop (1947-2007)

# ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for helping guide me throughout the duration of my PhD study. I'm thankful for all the efforts he took took help develop my various skills in systems experimentation and writing.

I would like to thank my committee members Prof. Sadayappan and Prof. Qin for their valuable guidance and suggestions.

I'm thankful to Lawrence Livermore National Laboratory for giving me a chance to see my research in action and see new areas of research through two internships. I'm especially grateful for Adam Moody and the help he extended to me while there.

I'm grateful to have had Sayantan Sur as a mentor during my first years of graduate study. Without him I would not have been able to make the progress that I have made. I'd also like to thank other senior colleagues of Nowlab including Lei Chai, Wei Huang, Amith Mamidala, Sundeep Narravula, Gopal Santhanaraman, and Karthik Vaidyanathan. I'm also thankful to have worked with many others in the lab including Jaidev Sridhar, Tejus Gangadharappa, Karthik Gopalakrishnan, Jonathan Perkins, Krishna Chaitanya, Ping Lai, Hari Subramoni, Greg Marsh, Xiangyong Ouyang and Ajay Sampat.

I would like to thank my family including my parents and my brother. They kept me going when going through some tough times. I'd also like to especially thank my fianceé Erin. She gave me support and a listening ear that was invaluable. I would not have had made it this far without their love and support.

# VITA

| | |
|---|---|
| September 6, 1982 | Born - San Diego, CA, USA. |
| August 2000 - May 2004 | B.C.S. Computer Science, Calvin College, Grand Rapids, Michigan. |
| June 2005 - August 2005 | Intern, Lucent Corporation, Columbus, OH. |
| September 2004 - December 2006 | Graduate Teaching Associate, The Ohio State University. |
| July 2006 - September 2006 | Summer Scholar, Lawrence Livermore National Laboratory, Livermore, CA. |
| June 2007 - September 2007 | Summer Scholar, Lawrence Livermore National Laboratory, Livermore, CA. |
| June 2008 - September 2008 | Research Intern, IBM T. J. Watson Research, Hawthorne, NY. |
| January 2006 - Present | Graduate Research Associate, The Ohio State University. |

# PUBLICATIONS

M. Koop, J. Sridhar and D. K. Panda, "TupleQ: Fully-Asynchronous and Zero-Copy MPI over InfiniBand", IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2009), Rome, Italy, May 2009

K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop and D. K. Panda, "Designing Multi-Leader-Based Allgather Algorithms for Multi-Core Clusters", 9th Workshop on Communication Architecture for Clusters (CAC 09), Rome, Italy, May 2009

J. Sridhar, M. Koop, J. Perkins and D. K. Panda, "ScELA: Scalable and Extensible Launching Architecture for Clusters", International Conference in High Performance Computing (HiPC08), Bangalore, India, December 2008

M. Koop, J. Sridhar and D. K. Panda, "Scalable MPI Design over InfiniBand using eXtended Reliable Connection", IEEE Int'l Conference on Cluster Computing (Cluster 2008), Tsukuba, Japan, September 2008

W. Huang, M. Koop and D. K. Panda, "Efficient One-Copy MPI Shared Memory Communication in Virtual Machines", IEEE Int'l Conference on Cluster Computing (Cluster 2008), Tsukuba, Japan, September 2008

R. Kumar, A. Mamidala, M. Koop, G. Santhanaraman and D. K. Panda, "Lock-free Asynchronous Rendezvous Design for MPI Point-to-point communication", EuroPVM/MPI 2008, Dublin, Ireland, September 2008

M. Koop, W. Huang, K. Gopalakrishnan and D. K. Panda, "Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand", 16th IEEE Int'l Symposium on Hot Interconnects (HotI16), Palo Alto, CA, August 2008

M. Koop, R. Kumar and D. K. Panda, "Can Software Reliability Outperform Hardware Reliability on High Performance Interconnects? A Case Study with MPI over InfiniBand", 22nd ACM International Conference on Supercomputing (ICS08), Island of Kos, Greece, June 2008

M. Koop, T. Jones and D. K. Panda, "MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand", IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS 2008), Miami, FL, April 2008

W. Huang, M. Koop, Q. Gao and D. K. Panda, "Virtual Machine Aware Communication Libraries for High Performance Computing", SuperComputing (SC07), Reno, NV, November 2007

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

Q. Gao, W. Huang, M. Koop and D. K. Panda, "Group-based Coordinated Checkpointing for MPI: A Case Study on InfiniBand", International Conference on Parallel Processing (ICPP07), XiAn, China, September 2007

S. Sur, M. Koop, L. Chai and D. K. Panda, "Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand Architecture with Multi-Core Platforms", 15th IEEE Int'l Symposium on Hot Interconnects (HotI15), Palo Alto, CA, 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 ACM International Conference on Supercomputing (ICS07), Seattle, WA, June 2007

W. Huang, J. Liu, M. Koop, B. Abali and D. K. Panda, "Nomad: Migrating OS-bypass Networks in Virtual Machines", 3rd International ACM Conference on Virtual Execution Environments (VEE07), San Diego, CA, June 2007

M. Koop, T. Jones and D. K. Panda, "Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach", 7th IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid07), Rio de Janeiro, Brazil, May 2007

A. Vishnu, M. Koop, A. Moody, A. Mamidala, S. Narravula and D. K. Panda, "Hot-Spot Avoidance With Multi-Pathing Over InfiniBand: An MPI Perspective", 7th IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid07), Rio de Janeiro, Brazil, May 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 (SC06), Tampa, FL, November 2006

M. Koop, W. Huang, A. Vishnu and D.K. Panda, "Memory Scalability Evaluation of the Next-Generation Intel Bensley Platform with InfiniBand", 14th IEEE Int'l Symposium on Hot Interconnects (HotI14), Palo Alto, CA, August 2006

## 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. F. Qin |

# TABLE OF CONTENTS

**Page**

# LIST OF TABLES

# LIST OF FIGURES

**Figure** **Page**

# CHAPTER 1

# INTRODUCTION

In the past decade, rapid advances have taken place in the field of computer and network design enabling us to connect tens of thousands of computers together to form high performance clusters. These clusters are used to solve computationally challenging scientific problems. These applications are wide ranging and include weather prediction, molecular dynamics, aircraft design and fusion simulations.

The Message Passing Interface (MPI) is a popular model to write applications for these clusters. It is a model that provides a way to conceptually "glue" all of the distinct computers into a single computational machine. Developed in the 1990s, this is now the defacto standard for parallel computing. MPI provides a portable abstraction for exchanging data between processes. 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 as well.

The amount of computational power needed continues to grow. For many problems an almost unlimited amount of computation can be needed. Even simulating a millisecond of a protein interaction cannot be performed by today's computers. The most powerful systems have only simulated up to a microsecond of this interaction [68]. As a result, applications continue to take advantage of increasing computational power. MPI is a key part of this as

these applications interact with the network through this library. 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 deliver higher performance.

InfiniBand [28] is a cluster interconnect which is based on open-standards and is gaining rapid acceptance. Originally designed as a generic I/O bus to connect graphics cards, storage, and CPUs, InfiniBand has become well accepted in the area of High Performance Computing (HPC).

As InfiniBand clusters continue to expand to ever increasing scales, the need for scalability and performance at these scales remains paramount. As an example, the "Ranger" system at the Texas Advanced Computing Center (TACC) includes over 60,000 cores with nearly 4000 InfiniBand ports [81]. By comparison, the first year an InfiniBand system appeared in the Top500 list of fastest supercomputers was in 2003 with a 128 node system at NCSA [2]. The latest list shows over 28% of systems are now using InfiniBand as the compute node interconnect. As a result, designing a scalable and high-performance MPI library for ultra-scale InfiniBand clusters is of critical importance.

This additional scale and the expected further growth of InfiniBand clusters is placing additional demands on the MPI library that were unthinkable even a few years ago. In this dissertation we examine the issues that arise in both current and future systems. Novel techniques are developed for reducing the memory footprint on ultra-scale systems and to increase performance. The transports of InfiniBand are studied in depth to determine the tradeoffs that each bring for performance and resource consumption.

The InfiniBand Architecture specification defines four different transports, however, all MPI designs over InfiniBand currently use only the Reliable Connection (RC) transport.

The RC transport is a reliable, connection-oriented transport and is also the most feature-rich transport of InfiniBand. When using the RC transport the network hardware provides reliability and message segmentation.

Using the RC transport, however, can have significant resource requirements as the number of processes in an MPI job increase. As a connection-oriented transport, each communicating peer must have a dedicated connection and associated memory resources of many KB per connection. This connection memory on earlier[1] designs can reach to be hundreds of MB per process at 8,192 MPI processes. Current MPI cluster sizes are already an order-of-magnitude higher than this level, showing an urgent need for scalability.

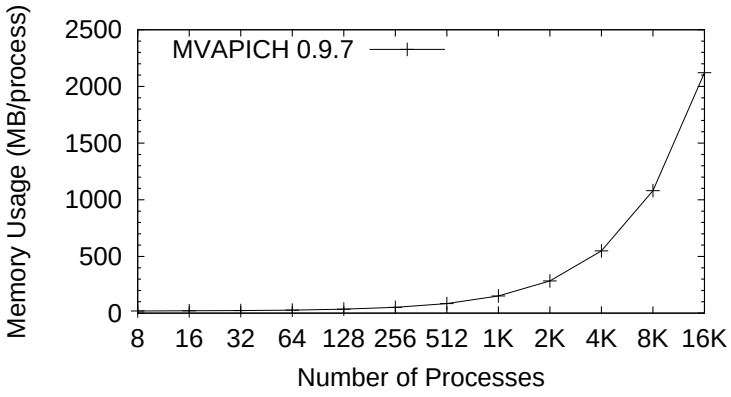



Figure 1.1: Memory Usage of MPI over InfiniBand (Earlier Design)

The other three original transports of InfiniBand are Reliable Datagram (RD), Unreliable Datagram (UD) and Unreliable Connection (UC). The RD transport of InfiniBand is not implemented in any currently available hardware and cannot be used for MPI. The UD transport is datagram-based and is the most scalable in resource usage since dedicated

[1]This refers to MVAPICH 0.9.7 from March 2006 when much of this work began

connections are not needed. The UC transport is connection-oriented like RC, but does not provide reliable or in-order service. Neither of these transports have been investigated in published work for MPI prior to this work.

Since the scalability problems of RC have become known another transport for InfiniBand has recently been introduced. The eXtended Reliable Connection (XRC) transport has the same features as RC, however, it provides a unique connection model that can save memory resources on multi-core machines. Prior to this work there also had been no studies on designing MPI for this transport.

In this dissertation a high-performance and scalable MPI library is designed for current-generation clusters as well as the next-generation ultra-scale clusters. We focus our attention on how the underlying transport choice effects the performance and scalability of the MPI libraries. In our work we investigate the four available transports for InfiniBand as well as methods to combine all of them into a hybrid design. We seek to address the following six questions in the proposed thesis:

- How can current MPI libraries that utilize Reliable Connection (RC) be optimized for lower memory utilization?
- How can MPI be designed over the Unreliable Datagram (UD) transport and what benefits and difficulties does it bring?
- How can reliability be designed for large-scale networks for each of the transports of InfiniBand and what are the consequences?
- What are the challenges associated with designing MPI over the new eXtended Reliable Connection (XRC) transport and what is the resulting performance and memory usage?

- How can an MPI library be designed to leverage all available InfiniBand transports simultaneously for the highest performance and scalability?

- How can an MPI library be designed to leverage the appropriate InfiniBand transport to achieve communication and computation overlap?

The rest of this dissertation is organized as follows: In Chapter 2 we discuss existing technologies which provide background for our work including InfiniBand, MPI and general network technologies. Chapter 3 describes in detail the problems that are addressed in this dissertation. Chapters 4-11 discuss the detailed approaches and results for these problems. Open-source software developed as part of this dissertation is described in Chapter 12. Chapter 13 provides the conclusion and possible future research directions.

# CHAPTER 2

# BACKGROUND

In this section the necessary background to InfiniBand and the earlier design of MVAPICH is provided. In addition, this section also describes the lower-level features and transports provided by InfiniBand on which this dissertation is designed upon.

## 2.1 InfiniBand Overview

The InfiniBand Architecture [28] (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 the host operating system for submitting work requests to the channel adapter and returning completion status. Figure 2.1 depicts the architecture of an InfiniBand network.

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

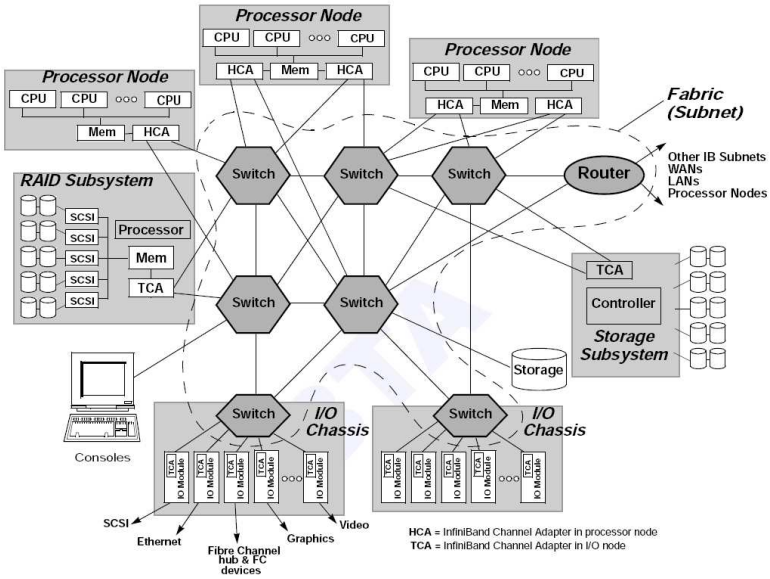



Figure 2.1: InfiniBand Architecture (Courtesy IBTA)

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 Work Queue Entries (WQEs) is reported through Completion Queues (CQ). Figure 2.2 shows a Queue Pair connecting two consumers and communication through the send and the receive queues.

### 2.1.1 Communication Semantics

InfiniBand supports two types of communication semantics: *channel* and *memory* semantics. In channel semantics, the sender and the receiver both must post work request entries (WQEs) to their QP. After the sender places the send work request, the hardware transfers the data in the corresponding memory region to the receiver end. It is to be noted

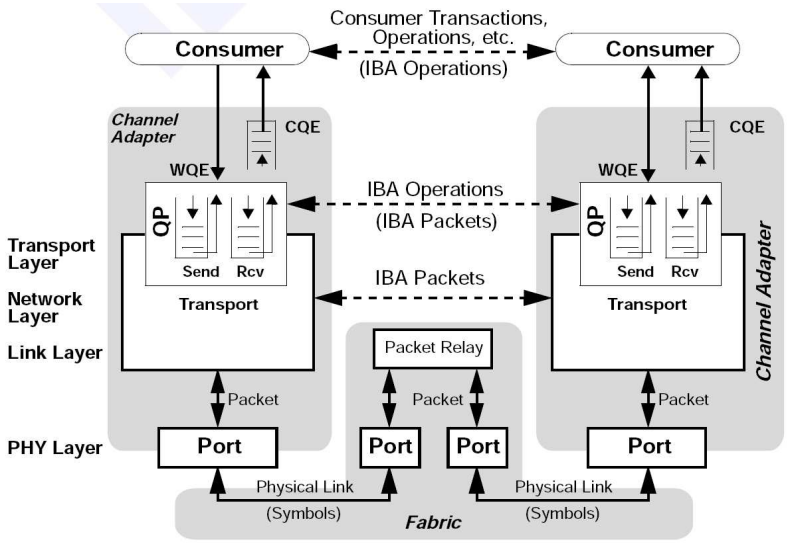



Figure 2.2: IBA Communication Stack (Courtesy IBTA)

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 [10], Quadrics [57] and others. The sender will not complete the work request until a receive request has been posted on the receiver. This allows for no buffering and zero-copy transfers.

When using channel semantics, the receive buffer size must be the same or greater than that of the sending side. Receive WQEs are consumed in the same order that they are posted. In the case of reliable transports, if a send operation is sent on a QP where the next receive WQE buffer size is smaller than needed the QPs on both ends of communication are put into the error state.

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 software involvement at the target. The remote host 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, although not all transports support it.

### 2.1.2 Transport Services

There are four transport modes defined by the InfiniBand specification: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC) and Unreliable Datagram (UD). Of these, RC and UD are required to be supported by Host Channel Adapters (HCAs) in the InfiniBand specification. RD is not available with any current hardware. All transports provide a checksum verification.

Reliable Connection (RC) is the most popular transport service for implementing services over InfiniBand. As a connection-oriented service, a QP with RC transport must be dedicated to communicating with only one other QP. A process that communicates with $N$ other peers must have at least $N$ QPs created. The RC transport provides almost all the features available in InfiniBand, most notably reliable send/receive, RDMA and atomic operations.

Unreliable Connection (UC) provides connection-oriented service with no guarantees of ordering or reliability. It does support RDMA write capabilities and sending messages larger than the Maximum Transfer Unit (MTU) of InfiniBand, which is 4KB. Being connection-oriented in nature, every communicating peer requires a separate QP. In regard to resources required, it is identical to RC, while not providing reliable service.

Reliable Datagram (RD) is a datagram-based transport. It provides most of the same features as RC, however, it does not require connection resources between nodes. There are no implementations of this transport on any current hardware.

| Attribute | RC | RD | UC | UD |
|---|---|---|---|---|
| Scalability (Number of QPs) | $n^2$ | $n$ | $n^2$ | $n$ |
| Corrupt Data Detected | Yes | Yes | Yes | Yes |
| Delivery Guarantee | Yes | Yes | No | No |
| Ordering Guarantee | Yes | Yes | No | No |
| Data Loss Detection | Yes | Yes | Yes/No | No |
| Error Recovery | Reliable | Reliable | Unreliable | Unreliable |
| RDMA Write | Yes | Yes | Yes | No |
| RDMA Read | Yes | Yes | No | No |
| Messages above MTU size | Yes | Yes | Yes | No |

Table 2.1: Comparison of InfiniBand Transport Types

Unreliable Datagram (UD) is a connection-less and unreliable transport, the most basic transport specified for InfiniBand. As a connection-less transport, a single UD QP can communicate with any number of other UD QPs. UD does have a number of limitations, however. Messages larger than an MTU size, which on current Mellanox [1] hardware is limited to 2KB, cannot be directly sent using UD. Only channel semantics are defined for UD, so RDMA is not supported. Additionally, UD does not guarantee reliability or message ordering.

Table 2.1 compares the various transports provided by InfiniBand. The connection-oriented transports (RC and UC) have additional features, such message segmentation and RDMA, but they require a QP per communicating process. The UD transport lacks those features, but has superior scalability. Also note that RC has delivery guarantees, which allows us to know if a transfer has failed.

### 2.1.3 Shared Receive Queue

Introduced in the InfiniBand 1.2 specification, Shared Receive Queues (SRQs) were added to help address scalability issues with InfiniBand memory usage. As noted earlier, in order to receive a message on a QP, a receive buffer must be posted in the Receive Queue (RQ) of that QP. To achieve high-performance MPI implementations pre-post buffers to the RQ to accommodate unexpected messages.

When using the RC transport of InfiniBand, one QP is required per communicating peer. To prepost receives on each QP, however, can have very high memory requirements for communication buffers. To give an example, consider a fully-connected MPI job of 1K processes. Each process in the job will require 1K - 1 QPs, each with $n$ buffers of size $s$ posted to it. Given a conservative setting of $n = 5$ and $s = 8KB$, over 40MB of memory per process would be required simply for communication buffers that may not be used. Given that current InfiniBand clusters now reach 60K processes, maximum memory usage would potentially be over 2GB per process in that configuration.

Recognizing that such buffers could be pooled, SRQ support was added so instead of connecting a QP to a dedicated RQ, buffers could be shared across QPs. In this method, a smaller pool can be allocated and then refilled as needed instead of pre-posting on each connection.

Note that a QP can only be associated with one SRQ for RC and UD. Thus, any channel traffic on a QP will consume a receive buffer from the attached SRQ. If another SRQ is desired instead, a second QP must be created.

### 2.1.4 eXtended Reliable Connection (XRC)

As noted in Section 2.1.2 the connection-oriented transports of InfiniBand require a QP per communicating process. On large-scale clusters this can use a significant amount of memory. In response to this, the eXtended Reliable Connection (XRC) transport was proposed by Mellanox in their ConnectX HCA. This allows a process to need only one QP to send a message to any process on another node. This can potentially reduce the number of QPs required by a factor equal to the number of processes per node.

This transport maintains the same functionality as the RC transport, but requires the use of SRQs. This new transport opens a new set of design possibilities.

### 2.1.5 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 (i.e. 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 physical 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.

### 2.1.6 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 and 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 needs 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 avoids unnecessary costs (like context-switch) associated with interrupts. This asynchronous mode is not used often in MPI since each MPI process is generally dedicated a core and polling gives higher performance.

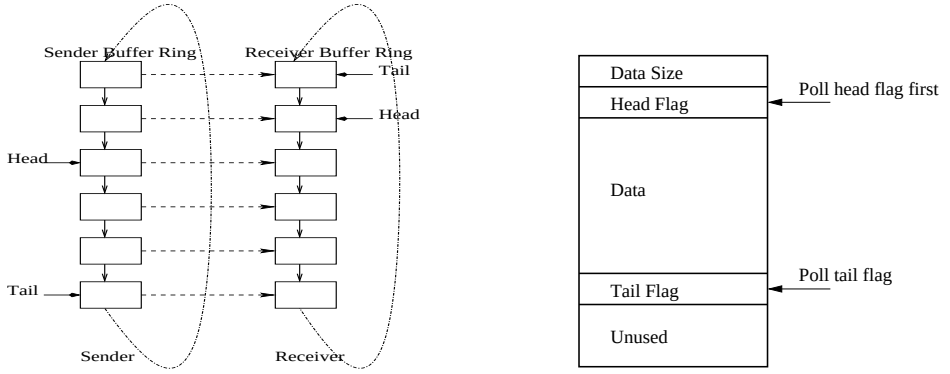
## 2.2 MVAPICH Design Overview

In this section we describe the design of MVAPICH [53] (MPI over InfiniBand). MVAPICH is based on the ADI2 interface of MPICH [24] and was derived from MVICH [34]. MVAPICH is currently in use at over 900 organizations around the world. It has helped several clusters achieve top rankings in the Top500 [2] list. The initial design and implementation of MVAPICH was done by Liu et. al at the Network-Based Computing Laboratory [35, 41, 31, 37, 40, 39]. In this section we will discuss several key aspects of the MVAPICH design which are pertinent to this dissertation. Particularly, this dissertation relates to the design of point-to-point communication of MVAPICH.

Most modern MPI implementations [20, 24] employ two major types of protocols for performing point-to-point message passing – the *eager* and the *rendezvous* protocols. We describe these protocols and the corresponding design used in MVAPICH to implement them:

### 2.2.1 Eager Protocol

The eager protocol is designed for MPI to provide low-latency and fast progress for small message sizes. In this protocol, small messages are simply sent to the receiver by the sender without any synchronization. MVAPICH has two main eager protocol designs based on the InfiniBand send/receive and RDMA semantics. They are:




(a) RDMA Channel Design

(b) RDMA Buffer Polling Mechanism

Figure 2.3: RDMA Channel Design in MVAPICH (Courtesy [41])

1. **Reliable Connection Fast-Path (RC-FP)**: InfiniBand adapters only reach their lowest latency when using RDMA write operations, with channel semantics having a 2$\mu$sec additional overhead (e.g. 5$\mu$sec vs. 3$\mu$sec) on our evaluation hardware. The

newest Mellanox adapter, ConnectX [47], reduces this gap to less than a microsecond, however RDMA write operations still achieve the lowest latency [77].

The RC-FP channel was first proposed by Liu et al [41] to leverage the low-latency of the RDMA write operation provided by InfiniBand. The design of this channel is shown in Figure 2.3(a). In this design, each sender/receiver process pair has a dedicated set of buffers associated with them. The association of these buffers is persistent. As noted in Section 2.1.1, the RDMA operations do not cause any completion event to be generated at the receiver end. To allow the receiver to discover the completion of packets sent over this RDMA channel, the sender places a particular flag at the memory location just following the end of the data buffer. This is shown in Figure 2.3(b).

The default MVAPICH configuration requires over 300KB of memory per RC-FP channel created. To limit memory usage, channels are currently setup adaptively and limited to a configurable number of channels in current MPIs over InfiniBand. In addition, each RC-FP channel requires polling an additional memory location for detection of message arrival. For example, communication with $n$ peers using the RC-FP channel requires polling $n$ memory locations for message arrival.

2. **Reliable Connection Send/Receive (RC-SR)**: This channel utilizes the InfiniBand Send/Receive semantics. Unlike the RC-FP channel described above, this does not provide the best point-to-point latency. However, this channel does provide some other features that are useful for large scale clusters. Each process can locally register and post receive requests. It is not even necessary for these receive regions to be contiguous in memory as in RC-FP.

It is the primary form of communication for small messages on nearly all MPI implementations over InfiniBand. Two designs have been proposed, one with per-peer credit-based flow control and the other using the Shared Receive Queue (SRQ) support of InfiniBand.

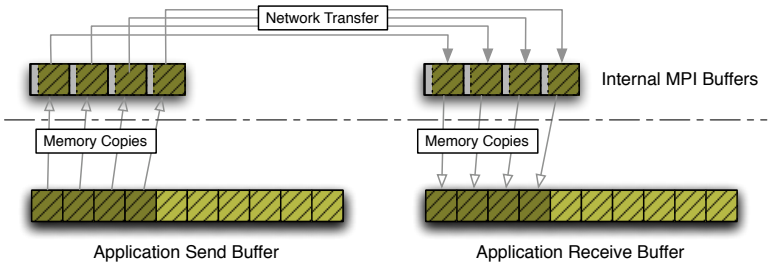
### 2.2.2 Rendezvous Protocol

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 [53, 51, 63].

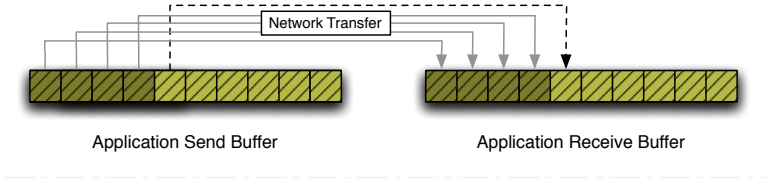
MVAPICH also has two main rendezvous protocol designs based on the InfiniBand send/receive and RDMA semantics. They are:

- **Reliable Connection RDMA (RC-RDMA)**: The MVAPICH design utilizes RDMA Write operations to eliminate intermediate message copies and efficiently transfer large messages in a "zero-copy protocol". 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 buffer information of the receiver along with the remote key to access that memory region. The sending process then sends the large message directly to the application buffer of the receiver 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.

- **Copy-Based Send**: Memory registration in InfiniBand is an expensive operation. For some messages above the eager threshold, but too small to efficiently register,




(a) Copy-Based




(b) Zero-Copy

Figure 2.4: Copy vs. Zero Copy

a copy-based approach is used. Large messages can be segmented within the MPI library into many small sends and sent using an eager protocol channel (after negotiating buffer availability). This method, however, introduces intermediate copies and degrades performance for large messages.

## 2.3 Zero-Copy Protocols and RDMA

Memory copies lead to overall degradation of application performance and ineffective use of resources of the computing platform. Memory copies become a bottleneck especially when transferring large messages. State-of-the-art MPI implementations over high-performance networks avoid memory copies for large messages by using zero-copy protocols. In zero copy protocols the sender and receiver use small control messages to match the message and then the message data is placed directly in user memory. A zero-copy protocol can significantly increase bandwidth and reduce cache pollution. Figure 2.4

shows the difference between a typical copy-based approach and a zero-copy approach. Instead of performing data copies in the send and receive paths, within user-space or the kernel, a zero-copy approach directly sends the data from the source application buffer to the final destination buffer.

Many zero-copy mechanisms have been implemented, including in U-Net [87], BIP [61], and PM [16]. Modern high-performance interconnects such as InfiniBand, Myrinet, and Quadrics use zero-copy communication. In order to support zero-copy messaging, the network and drivers must support OS-bypass to avoid kernel copies. There are two major ways zero-copy protocols are implemented in MPI implementations over modern interconnects:

- *Rendezvous Protocol using RDMA*: In this method, a handshake protocol (rendezvous) is used. The sending process sends a Request to Send (RTS) message with message tag information. Upon discovery of the message by the MPI layer at the receiver end, the receiver can either use RDMA Read to get the message data directly into user application buffer or send a Clear to Send (CTS) to the sender. If the sender receives a CTS, then it can use RDMA Write to send the message data directly to user application buffer at remote side. Thus, RDMA Read/Write provide a convenient method to perform zero-copy protocols when MPI message tags are matched by the MPI library. Typically, this method is used with networking stacks which do not have the capability to match MPI tags, e.g. InfiniBand/OpenFabrics [55].

- *Matched Queues Interface*: In this method, the sending process directly sends the message to a remote process. This requires that either the network device or network software layers be able to decipher the message tags from the sent message and match it with the posted receive operations. Upon a successful match, the rest of

the message may be directly received into the user application memory. This method is used in Myrinet/MX [50], InfiniPath/QLogic [62] and Quadrics [57].

## 2.4 Modern Network Technologies and Packet Loss

In this section we give background on network technologies used in modern networks and their effect on communication reliability.

*Link Level Flow Control*: Link level flow control is a critical feature of all networks [3, 21]. Many high-performance System Area Networks (SAN) provide link level credit flow control. These interconnects, such as ASI [46], InfiniBand, and Fibre Channel generally do not drop packets due to congestion [64]. As a result, packet loss is minimal. Traditional Ethernet, however, will drop packets due to congestion. In this case, packets do not always arrive at the receiver side.

*Cyclic Redundancy Checks (CRC)*: Modern networks provide a link-level CRC check to provide data reliability. Networks such as InfiniBand implement an end-to-end link-level CRC as well as per-hop CRC check. Other networks such as Myrinet [10] also provide CRC at the hardware level. Others have advocated for additional memory-to-memory CRC checks to prevent against I/O errors [30].

*Reliability*: Unlike many of the previous technologies, high-performance interconnects differ in their implementation of reliability. Myrinet requires software-level reliability, Quadrics [57] implements hardware-level reliability, and InfiniBand implements both modes of operation.

*Packet Loss*: Many modern network technologies implement link level flow control to prevent packet loss from occurring due to congestion. Additionally, many networks provide

a link-level CRC check, which represents one of the few times a network will drop a packet. Empirical data confirms these very low drop rates.

# CHAPTER 3

# PROBLEM STATEMENT AND METHODOLOGY

The main objective of this dissertation is to explore the various InfiniBand transports and how they can be used within the MPI library to enhance performance, scalability, and reliability. New protocols and designs for the InfiniBand transports are designed. In many cases these transports have never been used for MPI before, and others that have been used in MPI before we further optimize.

Figure 3.1 shows the various components of our proposed research framework. In general, this dissertation focuses on the shaded boxes. At the lowest layer we optimize the Reliable Connection (RC) connections and design message passing over the Unreliable Datagram (UD), Unreliable Connection (UC) and eXtended Reliable Connection (XRC) transports. To support these designs we design a "Reliability and Reordering Protocols" layer. This allows reliable and efficient data reliability, despite the lower layers potentially being unreliable. Further, we develop a "Message and Channel Scheduling" layer that can dynamically schedule messages on different transports and protocols according to application patterns and resource usage. Additionally, this dissertation focuses on providing fault tolerance for large-scale clusters and allowing communication-computation overlap that is key for ultra-scale applications. All of these designs are part of the MPI library, which allows end user applications to benefit directly without modification.

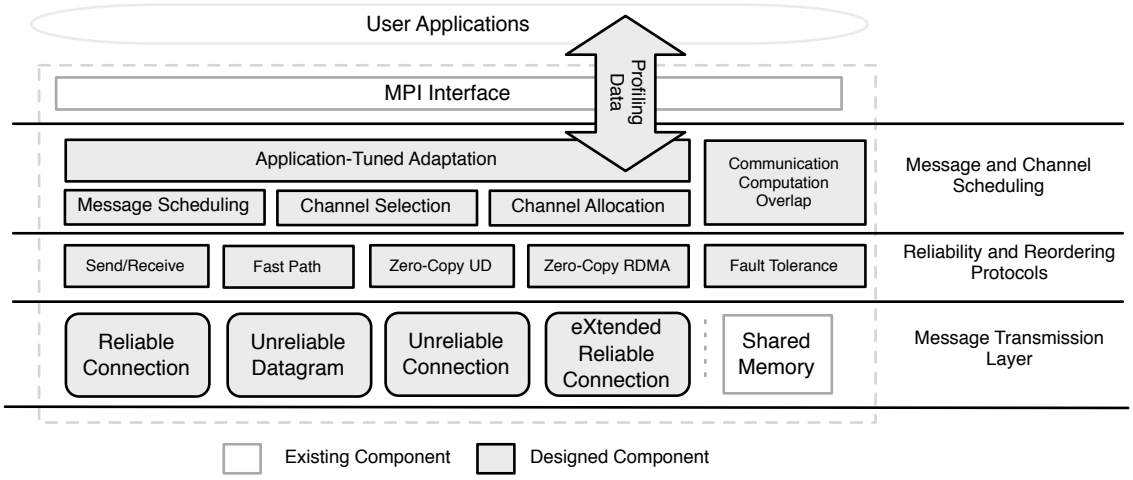



Figure 3.1: Research Framework

Using this framework we aim to address the following questions:

- **How can current MPI libraries that utilize Reliable Connection (RC) be optimized for lower memory utilization?**

  Current MPI libraries over InfiniBand use the RC transport as the primary transport mode for message passing. As described in Section 2.1, as a connection-oriented transport, RC requires a dedicated connection resource for every communicating process in the job. Previous work [69, 75] has reduced this memory with the Shared Receive Queue (SRQ) feature of InfiniBand, but significant resources are still required. As part of this study we show that memory consumption can grow up to 1 GB/per process at only 8K processes.

  In Chapter 4 methods to reduce this memory consumption are examined. A key factor in memory usage is found to be the number of Work Request Entries (WQEs)

allocated per connection. The number of WQEs denotes the number of simultaneously outstanding send operations allowed at a single time. Simply reducing this value, however, can cause severe performance degradation. To address this concern a message coalescing technique is proposed to reduce the number of outstanding send operations required to maintain performance.

Second, we find that the method that MPI libraries use to transfer small messages (RC-FP) can use a significant amount of memory. In Chapter 10 a new design for small message transfer over RDMA is proposed and evaluated.

- **How can MPI be designed over the Unreliable Datagram (UD) transport and what benefits and difficulties does it bring?**

  Despite optimizations in memory usage for the RC transport, memory consumption for connections can still reach hundreds of MB per process at larger scales. To address this concern an MPI is designed over the connection-less UD transport of InfiniBand in Chapter 5. As a datagram transport, only a single communication endpoint is required per process. This allows superior scalability for ultra-scale clusters.

  There are a number of issues with the UD transport that require a careful design study to map it to MPI efficiently though. The UD transport provides no reliability or ordering guarantees. Additionally, it provides no RDMA or message segmentation support.

  Additionally, the UD transport has very poor bandwidth for applications due to the lack of RDMA and the need to provide reliability in software. To address this we propose a novel zero-copy transfer mechanism over UD that can increase performance to near that of RC.

- **How can reliability be designed for large-scale networks for each of the transports of InfiniBand and what are the consequences?**

  In Chapter 6 the unreliable transports of InfiniBand are examined to determine the cost of providing reliability. The Unreliable Connection transport has not previously been used for MPI. Thus, a new MPI design is proposed over UC and tradeoffs in the design are explored. More importantly, this design is also a tool to quantitatively analyze the cost of providing reliability in hardware. Since the significant difference between RC and UC is the lack of hardware reliability, this design allows a comparison of the costs of hardware versus software reliability.

  Large-scale clusters also face the increasing problem of hardware failures as the number of components increases. Switches can break or reboot, cables can be faulty, or even the end HCAs can fail. In Chapter 11 a new reliability framework is designed to allow MPI jobs to survive these types of failures. The performance is also evaluated to show a very low overhead despite this added reliability.

- **What are the challenges associated with designing MPI over the new eXtended Reliable Connection (XRC) transport and what is the resulting performance and memory usage?**

  Recently, in response to the increased memory usage for RC communication contexts, the XRC transport has been introduced for InfiniBand. It adds additional capabilities that allow connections to be shared between processes on the same node, but preserves the features of RC.

  In Chapter 8 various MPI designs are proposed for this transport. This is done to analyze the design issues and tradeoffs involved in mapping MPI to this new transport.

In particular, the connection mapping is now on a process to node basis rather than a process to process basis, however, connections are not symmetric. This leads to additional connection setup design choices that must be evaluated carefully. Performance and memory scalability is evaluated for this mode.

- **How can an MPI library be designed to leverage all available InfiniBand transports simultaneously for the highest performance and scalability?**

  From our earlier studies on the various InfiniBand transports available, we have shown that each transport provides different characteristics in performance and resource scalability. The key component of this dissertation is to investigate how all of these transports can be combined within a single MPI design to optimize the overall application execution.

  In Chapter 7 a new MPI library design termed "Aptus" is proposed that can simultaneously use different message transfer protocols and transports. This framework can allow superior scalability as well as the highest performance since a transport can be used only for communication patterns where it will perform the best. We leverage the results from the MPI designs on each transport to design the adaptive messaging techniques.

- **How can an MPI library be designed to leverage the appropriate InfiniBand transport features to achieve communication and computation overlap?**

  Communication and computation overlap is a key concern for large-scale systems as the number of processes increase. There is a need to make the communication less dependent on synchronization and allow communication to proceed even when computation is being performed.

In Chapter 9 a new design for allowing full overlap of communication and computation is proposed. It uses the new XRC transport of InfiniBand to allow the hardware to perform the task of all matching operations.

# CHAPTER 4

# REDUCING MEMORY FOR RELIABLE CONNECTION WITH MESSAGE COALESCING

In this chapter, the memory scalability issues for MPI and other applications that make use of the Reliable Connection (RC) transport of InfiniBand are explored. In particular, we evaluate the memory usage of MVAPICH [42, 52], a popular MPI implementation over InfiniBand, and quantify the effect of the number of allowed outstanding send operations on memory usage. As part of this study we propose reducing the number of allowed outstanding send operations and a coalescing method to eliminate the resulting performance degradation. While MPI is examined in-depth, this same analysis and design can apply to other applications or protocols that make use of InfiniBand connections.

This new design reduces memory usage at 8,192 processes by an order of magnitude and maintains performance equivalent to the existing design. Our results show an increase in small message performance of up to 150% and near identical performance for message sizes above that level. We additionally validate our design with the NAS Parallel Benchmarks, sPPM, SMG2000, and Sweep3D and note that performance remains unchanged despite reducing memory usage significantly.

The rest of the chapter is organized as follows. In Section 4.1 we provide an evaluation of the memory usage by the MPI library with different numbers of allowed outstanding

sends. Section 4.2 describes our design of coalescing packets when many small messages are sent within a short time period. Our enhanced design is evaluated in Section 4.3. Related work is covered in Section 4.4.

## 4.1 Work Queue Entries and Memory Usage

In this section we discuss the purpose and memory usage of send Work Queue Entries (WQEs). In particular, we examine memory usage when all connections are made between processes in the cluster.

### 4.1.1 Memory Usage with Varied Numbers of WQEs

To observe the memory usage of the MPI library, we run an MPI application with the static connection settings of MVAPICH and measure the total memory usage. The average value per process is reported. To better assess only the InfiniBand connection costs, we disable shared memory support. Even when shared memory communication is used, however, the per connection values we report are accurate and the memory usage at large numbers of processes is nearly identical to the case without shared memory support.

Figure 4.1 shows the memory usage per process with increasing numbers of processes and varied allocations of send WQEs. Results are experimentally obtained through 1024 processes; numbers above that level are modeled. We observe that for 8192 processes, using the default allocation of 200 send WQEs requires around 1 GB of memory per process. On a machine with 8 cores per node, such as the Peloton clusters at Lawrence Livermore National Laboratory (LLNL), a total of 8 GB per node would be used only for the MPI library with this allocation. With 16 GB per node, 50% of available memory is consumed only for connection memory. When the number of send WQEs per connection is decreased to 5, the memory usage drops to less than 90 MB per process at 8192 processes.

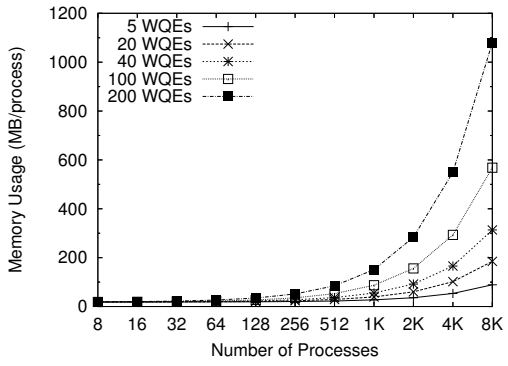



Figure 4.1: Memory Usage (per process) with Varied Numbers of WQEs

Table 4.1: Memory usage per additional connection with increasing send WQEs

| 5 | 10 | 20 | 40 | 100 | 200 |
|---|---|---|---|---|---|
| 8.82 KB | 12.82 KB | 20.81 KB | 36.86 KB | 68.73 KB | 132.76 KB |

Table 4.1 shows the additional memory required per connection based on the experimental evaluation up to 1024 processes. From this information we can determine that there are approximately 4 KB of connection costs other than send WQEs, but after 5 send WQEs per QP that term becomes dominant.

It should be noted that InfiniBand allows for different sizes of *inline* data. This allows the reduction of latency by $\sim$ 200 nanoseconds by storing the data, generally up to a maximum of $\sim$ 200 bytes (HCA dependent), in the request. Since inline data is stored within each WQE, it increases the size of the data structure. In the version of the InfiniBand drivers used for this work, the memory usage remained unchanged for different values of

inline data. Newer versions of these drivers have reduced memory when smaller inline limits are used in QP creation.

### 4.1.2 Effects of Reducing WQEs

As noted in the previous subsection, the MPI library memory usage is highly dependent on the memory allocation of the QP and WQE resources. Reducing the number of send WQEs leads to a large decrease in connection memory usage, however, there are performance consequences to lowering the number of available WQEs.

As noted in Section 2.1, the number of send WQEs restrict the number of send operations that can be outstanding on a given QP. In this section we evaluate the issues with lowering the number of send WQEs. The default value of 200 was empirically derived so buffering in the MPI library is extremely rare.

Results from throughput microbenchmarks show that small message throughput is diminished considerably when the number of send WQEs is reduced. Figure 4.2 shows that performance drops over 70% from nearly 900,000 to under 200,000 messages per second. From Figure 4.3 we observe the overall data throughput is also reduced significantly for small to medium-sized messages.

It is important to note that with these throughput microbenchmarks the sending process sends a window of 64 MPI messages before an acknowledgment is sent by the receiver. This process is repeated many times to get an average value. Thus, the performance degrades since the send WQEs are exhausted before the window is complete and additional sends cannot be pipelined by the HCA. This means that the bandwidth for a single message is the same in the case of both 200 and 5 WQEs, however, if messages are sent in bursts

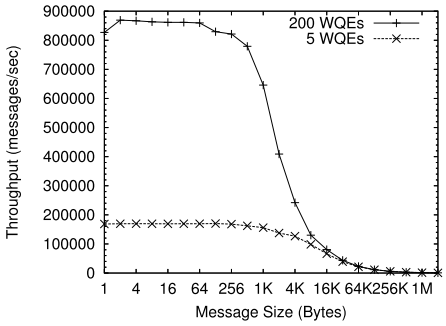



Figure 4.2: Uni-Directional Messaging Rate with Existing Design

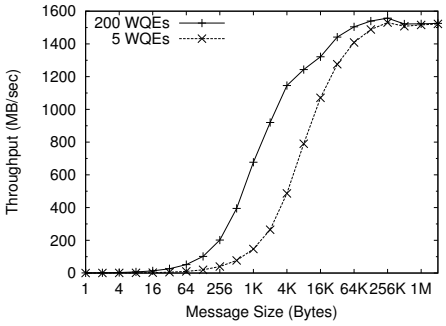



Figure 4.3: Uni-Directional Bandwidth with Existing Design

then the performance will be degraded when fewer WQEs have been allocated. Regardless, it is not possible to simply decrease the number of WQEs, doing so has performance consequences.

## 4.2 Message Coalescing Design

In this section we describe our design to alleviate any issues that may arise as a result of reducing the number of send WQEs available to the MPI library.

### 4.2.1 Motivation

As mentioned in Section 2.1, a send WQE is needed to issue a send request to the HCA. If one is not available, the message must be queued internally in the library until a previous send operation completes. If the number of outstanding sends is to be reduced without sacrificing performance, a method is needed to efficiently send the messages that may be in the queue as quickly as possible.

From Figures 4.2 and 4.3 we observe that when the number of WQEs is low, small message performance is degraded significantly. With larger messages, the bandwidth can be saturated with only a few simultaneous sends. Since this is the case, our attention will focus on increasing the performance for messages up to 8K, where the performance gap is the largest.

The key issue is that the network fabric is not being efficiently utilized when only a few WQEs are available. In this case the HCA is not able to pipeline the requests optimally. To make better use of the network fabric we propose a scheme to coalesce the messages that are being queued to increase the network performance since there is a startup cost associated with each send operation. By coalescing the messages we are able to amortize the startup cost of the operation across the number of messages that we are able to pack together.

### 4.2.2 Design

As discussed in Section 2.1, memory used in communication must be registered. Since memory registration has a high startup cost, small messages are copied into pre-registered communication buffers before being sent in MVAPICH. The general send flow for a small message send operation therefore is to copy the contents of the user buffer into an internal registered MPI buffer, at which point the library determines if there are available send WQEs. If at least one WQE is available the message is immediately posted to the QP and sent by the HCA, otherwise it is placed on a linked-list queue. When completion of previous send operations are detected through the CQ, messages are sent in order from the queue.

To efficiently coalesce messages, we alter the send flow operation. Before copying the contents of the user buffer into a registered buffer we check to the availability of send WQEs. If there is availability, the flow continues as in the original case. If not, we first check the queue for any other messages waiting to be sent on that QP. Assuming another message is present we verify it is of a compatible type – we do not combine sends with rendezvous data messages or other special message types. If enough space is available in the communication buffer the message is coalesced and copied into the buffer. If there are no other pending sends for that QP or there is insufficient buffer space available, a new communication buffer is used and the message is copied into it. Note that we are not using any additional memory to coalesce the messages; we are potentially using less since we can use the same buffer for more than one message.

Another design alternative is to use the InfiniBand scatter/gather capabilities instead of packing into the same buffer. This, however, introduces an unnecessary overhead. Since the user buffer is already being copied into pre-registered buffers it is more efficient to coalesce into a buffer initially. In this way we only have one copy on the sender side as well.

To further optimize the throughput of messages we cache the MPI tag matching information for each message. If the tag information of a message matches that of the previous coalesced message, a one byte `COALESCED_CACHED` flag is set and the header information is omitted. Otherwise the entire header information is included in the coalesced message. This caching increases performance by nearly 10% for messages less than 64 bytes and is negligible for messages with payloads above 512 bytes. Thus, even without caching the coalescing design has performance significantly higher than that of the original design with 200 WQEs for these message sizes. In this way we achieve a general solution with

no requirement that all coalesced messages need to have the same tag matching information (size, tag, context), while obtaining optimal performance for those with identical tag information.

## 4.3 Evaluation

Our experimental testbed is a 575-node InfiniBand Linux cluster at Lawrence Livermore National Laboratory. Each compute node has four 2.5 GHz Opteron 8216 dual-core processors for a total of 8 cores. Total memory per node is 16GB. Each node has a Mellanox MT25208 DDR HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55]. The Intel v9.0 compiler is used for compilation of the MVAPICH library and applications.

### 4.3.1 Microbenchmark Performance

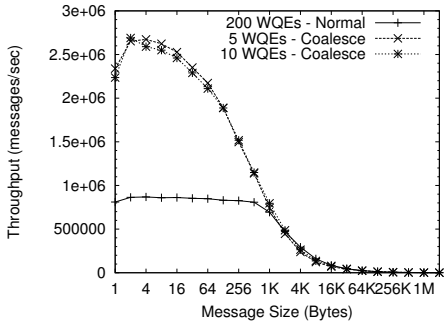



Figure 4.4: Coalesced Message Rate

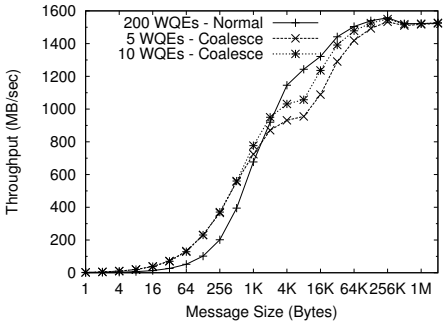



Figure 4.5: Coalesced Uni-Directional Bandwidth

Figure 4.4 shows that small message throughput for the coalesced design with 5 and 10 send WQEs significantly exceeds that of the existing design for message sizes through 1

KB. This is an effect of the amortization of startup costs for posting a send in InfiniBand. By packing more messages per send buffer we are able to lower the average overhead for sending a message. In this regard we have more than succeeded in removing the performance degradation that otherwise occurs with such a small number of send WQEs.

This finding is also seen in Figure 4.5, where we observe improved performance for messages up to 1 KB and slightly lower performance for messages from 4 KB to 128 KB. Performance has been improved significantly for small to medium messages, but the coalescing scheme cannot pack messages that exceed the half of the buffer space. The copy-based send method is used through 9 KB on this platform. The number of WQEs is important; using only 5 WQEs for the coalescing design cannot match the microbenchmark performance of 200 WQEs at all message sizes, however, 10 WQEs nearly matches the performance. Adding additional WQEs can close the performance gap.

Even higher messaging rates could likely be achieved with an even smaller number of send WQEs, or artificially restricting the number of outstanding send operations for small messages and then allowing messages greater than 2 KB to use additional WQEs. This is outside the scope of this work; we instead wish to find the lowest memory usage possible without harming performance rather than just increasing the message rate.

It is important to note that microbenchmark performance represents the extreme situation for a low number of send WQEs. This performance will only be seen when there are more outstanding sends being used by the application than available send WQEs.

Applications that do not use more than the available send WQEs or do so infrequently will perform equally with the coalescing design or the existing design. Differences in the microbenchmark graphs are not applicable unless the send WQEs are exhausted, as they are here. In this work we present these numbers since they represent the worst-case and even

in this case we are able to show near equal performance using only 15% of the memory usage.

### 4.3.2 Application Performance

We compare the performance of the existing design with the default number of WQEs with our proposed coalescing design using a low number of existing WQEs and significantly less memory. We evaluate with the NAS Parallel Benchmarks, and three applications from the ASC Benchmark Suite [4]: sPPM, SMG2000, and Sweep3D. Communication pattern analysis of sPPM, SMG200, and Sweep3D is available in [84] by Vetter, et al.

**Applications**

- **NAS Parallel Benchmarks** [8] are kernels designed to be typical of various MPI applications. As such, they are a good tool to evaluate the performance of the MPI library and parallel machines.

- **sPPM** [49] is an application distributed as part of the ASC Purple Benchmarks. It solves a 3D gas dynamics problem on a uniform Cartesian mesh using the Piecewise Parabolic Method (PPM).

- **SMG2000** [15] is a parallel semicoarsening multigrid solver, which is also a part of the ASC Purple Benchmarks.

- **Sweep3D** [25, 32] uses a multidimensional wavefront algorithm to solve the 3D, time-independent, particle transport equation on an orthogonal mesh.

### Results

All results are taken with 256 processes, 4 processes per node with shared memory support disabled to stress network performance. All application results are the average of 5 runs, although deviation in performance was low between runs.

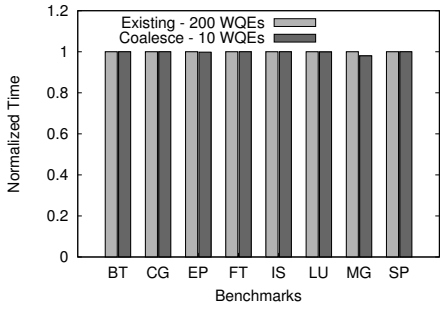



Figure 4.6: NAS Benchmarks Comparison, Class C, 256 Processes

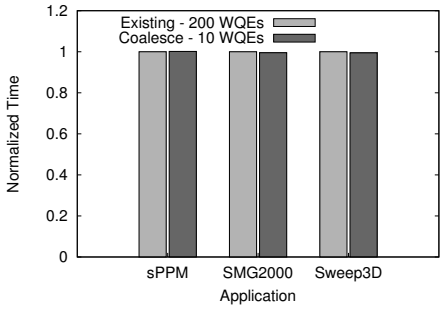



Figure 4.7: sPPM, SMG2000, and Sweep3D Results, 256 Processes

Figure 4.6 shows the performance of the existing and proposed designs with performance normalized to the existing design for all of the NAS Benchmarks, Class C. As expected from the microbenchmark results, there is no observed performance degradation. MG shows some benefits using our design over the original, although additional study is required to determine the cause of this improvement.

The performance comparison for sPPM, SMG2000, and Sweep3D is shown in Figure 4.7. Again we observe no performance degradation for our enhanced design. All application results are within one percent of each other.

As expected from the microbenchmark results, the performance of the proposed design is no worse in any application or benchmark than the existing design, despite using nearly an order of magnitude less memory.

## 4.4 Related Work

Memory usage of the MPI library has also been studied by many other researchers. Early versions of MVAPICH exhibited significant memory usage as the number of connections increased as studied by Liu, et al in [36]. Followup work by Sur, et al in [75] significantly reduced the per connection memory usage in MVAPICH using InfiniBand Shared Receive Queue (SRQ) support and a unique flow control method. Similar techniques have been used in Open MPI [20] by Shipman, et al in [69]. In both of these studies, the memory usage was reduced mostly through a reduction in communication memory buffer usage. In this work we have instead targeted the connection memory usage which remains a significant issue at scale. Adaptive connection management in MVAPICH to setup only those connections that are used was discussed by Yu, et al in [90]. This paper described a method to setup connections dynamically as needed. The methods we describe in this work can be used in conjunction with an on-demand strategy for optimal memory usage.

Our contribution with this work is evaluating the memory connection usage and the impact of send WQEs and proposing a coalescing method to maintain optimal performance using significantly less WQEs. Our design seeks to be comprehensive in providing near-identical performance using significantly less memory resources without imposing any restrictions on application behavior, such as the number of outstanding sends an application may have, the number of connections created, or their message tags.

# CHAPTER 5

# SCALABLE MPI OVER UNRELIABLE DATAGRAM

State-of-the-art MPI implementations over InfiniBand primarily use the Reliable Connection (RC) transport of InfiniBand since it is the most feature-rich – supporting high-throughput, reliability, atomic operations, and operations for zero-copy transfers of large buffers. The popular open-source MPI implementations over InfiniBand, MVAPICH [42] and Open MPI [20], use this transport layer for communication. As such, research studies on MPI implementations over InfiniBand have focused on managing resources for implementations using the RC transport. Strategies such as lazy connection setup and using the Shared Receive Queue (SRQ) support of InfiniBand have been employed to reduce resource consumption [90, 75, 69].

Using the RC transport, however, has the drawback that the worst-case memory usage per process of the MPI library across the cluster increases linearly with increasing processes. Since each connection requires several KB of memory and clusters continue to scale to tens of thousands of processors and above, the connection-less Unreliable Datagram (UD) transport of InfiniBand is a potentially attractive alternative. As a connection-less transport, the maximum memory used for connections across the cluster is static even as the number of processes increases. Using the UD transport for an MPI library, however, brings several challenges, including providing reliability. Additionally, the UD transport

limits the maximum message size to the Message Transfer Unit (MTU) size and lacks support for RDMA. The MTU on current Mellanox [1] hardware is 2KB.

In this chapter we analyze the design alternatives of the MPI library over the UD transport of InfiniBand. We propose three messaging protocols that provide reliability without sacrificing performance, and while quickly reacting to the expected number of packet drops on high-performance networks. We prototype our design and compare the performance and resource usage of our prototype versus the RC-based MVAPICH. We evaluate the NAS Parallel Benchmarks, SMG2000, Sweep3D, and sPPM up to 4K processes on an 1152-node InfiniBand cluster. For SMG2000, our prototype shows up to a 60% speedup and sevenfold reduction in memory for 4K processes. Additionally, based on an analytical model, our design has a potential 30 times reduction in memory usage over MVAPICH at 16K processes. To the best of our knowledge, this is the first research work that presents a high performance MPI design over InfiniBand that is completely based on UD and can achieve near identical or better application performance as compared to RC. We also further extend this work to include an additional zero-copy mode for transferring messages over UD, which can further increase performance for large messages.

The rest of the chapter is organized as follows: Further motivation is provided in Section 5.1. Our UD-based MPI designs are described in Section 5.2. The prototype implementation of the design is described in Section 5.3. We evaluate these designs in Section 5.4. We also propose an additional optimization for large messages by designing a novel zero-copy protocol using a "serialized" communication mode in Section 5.5. We evaluate this enhanced design in Section 5.6.

## 5.1 Motivation

In this section we will describe the factors motivating this work, including resource scalability as well as potential performance improvements that can be achieved with using the UD transport.

### 5.1.1 Resource Scalability

In this section we examine the memory usage of the MPI library and the resource requirements associated with a connection-oriented transport. We first classify the memory usage of the MPI library into three main categories and then examine each with regards to RC and UD.

- **Data Structure Memory**: Even when a connection is not created, memory allocations for data structures increase with job size.
- **Communication Context / QP Memory**: The memory required for QP contexts we refer to as the communication context memory. This memory requirement is the focus of this work.
- **Communication Buffer Memory**: Registered memory that is used for send or receive operations. This does not include user buffers that are registered to provide zero-copy communication, only those used for eager or packetized communication.

#### Data Structure Memory

Regardless of the transport type, the memory allocated for data structures usually grows with the number of total processes in a job. Although this memory usage per process grows linearly with increasing numbers of processes, the coefficient is quite low, generally much less than 1 KB.

### Communication Context / QP Memory

Communication context memory is the memory required for QP contexts. With the RC transport a separate dedicated QP must be created for each communicating peer. Unlike the memory required for data structures, the memory required for each additional QP is considerably greater. The resources required for a single QP context, as created with the default settings of MVAPICH 0.9.8, consume nearly 68 KB of physical memory.

In an attempt to minimize this memory, current MPI implementations over InfiniBand, including MVAPICH and Open MPI, employ an on-demand [89, 90] connection setup method to only setup connections and QPs as required. Thus, if an application communicates with only a few peers the number of connections and QPs required per process may be considerably fewer than the number of total processes in a job. This method, however, is only beneficial when the number of communicating peers is low. If an MPI application communicates with more than a small number of peers, memory usage will grow with the total number of processes in the job, potentially reaching above 1GB of memory per process for 16K processes. In [89, 84], SMG2000 [15] is found to have up to the same number of connections per process as total processes. With increasing scale the MPI library should be able to maintain a reasonable memory footprint regardless of the number of communicating peers.

Using UD as a transport, however, solves this significant problem of QP memory increasing with the number of peers. Since a single UD QP can communicate with any number of other UD QPs each MPI process need only allocate a single QP. Thus, as the number of communicating peers increases the connection memory remains constant. This reduction in connection memory can significantly increase the amount of memory available for the application.

#### Communication Buffer Memory

As mentioned in Section 2.1.1, to receive a message using channel semantics (send/receive), the receiver must post a receive buffer to the associated QP. To maintain high-performance, buffers are generally pre-posted to each QP; posting buffers per QP for a connection-oriented transport, like RC, however, requires significant memory usage. This prompted the InfiniBand specification to be updated to include support for Shared Receive Queues (SRQs), which allow receive buffers to be shared across QPs. Thus, a single pool of posted receives can be used for any number of QPs and MPI peers.

UD can also make use of an SRQ; however, it is not necessary since all receives can be posted directly to the single UD QP. Thus, for both RC and UD, the communication buffer memory can be allocated as a pool and does not depend directly on the number of processes in a job.

### 5.1.2 Performance Scalability

Performance can also potentially be improved using a connection-less and unreliable transport through better HCA cache and fabric utilization.

#### InfiniBand Context Memory Caching

InfiniBand HCAs cache communication context information using on-card memory, called the InfiniBand Context Memory (ICM) cache. The ICM cache has a limited size and cannot hold more than a limited number of QP entries at any one time; context information outside of the cache must be fetched from host memory. Sur, et al. in [79] provide an analysis of the Mellanox MT25218 HCA show that less than 128 QPs can be stored in the cache at any one time. Furthermore, the authors show that when a QP is not in the ICM

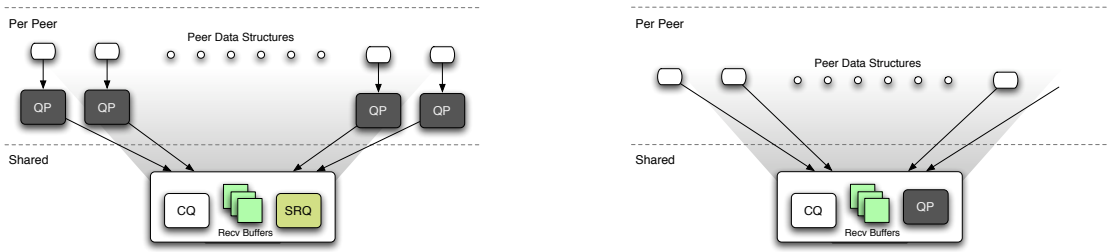
cache there is a significant latency penalty; for transfers less than 1 KB the latency nearly doubles. This is a significant problem, especially considering multi-core machines where all processes on a node will be sharing the cache. Using a connection-less transport, a single QP can communicate with many peer QPs, avoiding cache thrashing when communicating with many other QPs.

#### Fabric Utilization

When using a reliable transport, the HCA must provide reliability to guarantee that packets are not lost and are delivered in order. Providing this service at the transport layer instead of the application layer does not allow the application to optimize for out-of-order messages. For example, in MPI it is not necessary for all parts of a large message transfers to arrive in order. Instead, they can be placed in the user buffer as they arrive and not dropped by the HCA. Additionally, the lack of acknowledgements (ACKs) at the transport layer means less traffic overhead on the InfiniBand fabric. The upper layer can send ACKs in a more lazy method or another optimized way depending on application needs.

## 5.2 Design Alternatives

Providing a high-performance and scalable MPI over UD requires a careful design since many features, including reliability, lack of RDMA, and a small MTU, are provided by hardware when using the RC transport are now not available or must be done in the MPI library.




(a) MVAPICH (b) UD-based Design

Figure 5.1: Resource Allocation

### 5.2.1 Overview

Our UD-based design is different from MVAPICH and Open MPI in that it uses UD QPs for data transfer rather than RC. Using the connection-less UD transport allows much better memory scalability for large-scale clusters.

Figure 5.1 shows the resource allocations of our proposed design versus the design used in MVAPICH. In MVAPICH each peer requires a data structure and optionally a QP, created when communication is initiated during application execution. In the worst-case a QP will be allocated for each peer in this situation. Our UD-based design by contrast uses a single UD QP for all peers since it is a connection-less transport. In both designs the CQ and communication buffers are shared across all connections.

### 5.2.2 Reliability

A key issue that must be addressed when using an unreliable transport layer is how to provide reliability in a lightweight and scalable form. Significant prior work has been done on methods to provide reliable service at the transport layer; however, in-depth study has

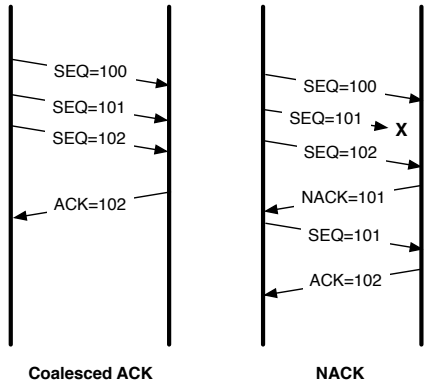



Figure 5.2: Coalesced ACK and NACK Protocols

not been done at the MPI library layer. LA-MPI [23, 6] has explored reliability against loss and corruption in the MPI library, but was more focused on I/O bus errors.

Our design uses the traditional sliding window protocol. The sender issues packets in order as there is available space in the window. In this manner the window represents the maximum number of unacknowledged message segments outstanding. Additional send operations occurring when the send window is already full are queued until outstanding operations have been acknowledged.

At the time of each send operation, a timestamp is associated with each message segment. To maintain high-performance, we use the Read Time Stamp Counter (RDTSC) assembly instruction to read the hardware counters for these values. If an ACK has not been received within a given timeout the segment is retransmitted.

**Optimizations**

We additionally use a negative acknowledgment (NACK) based protocol to request selective retransmissions. Upon receiving a message out of order we assume the skipped messages have been lost and request re-transmission from the sender. Using this mechanism we can handle potential message drops without having to wait for timer expiration.

We also borrow the traditional ACK coalescing method from TCP to reduce the number of acknowledgments.

**Reliability Progress Methods**

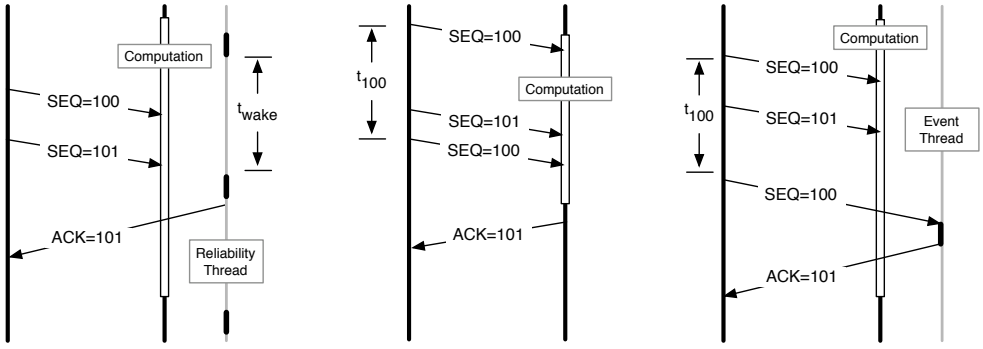
The main challenge in providing reliability at the user-level within the MPI library is the potential lack of progress. In a traditional sliding window sender retransmission protocol, message acknowledgments must be sent within a timeout period otherwise the message is retransmitted from the sender. There are two issues to consider:

- Sender Retransmission: How should the sender know when to retransmit a message? Should there be a timer-based interrupt or a thread?
- Receiver Acknowledgment: How can the receiver acknowledge the message within a bounded amount of time?

While providing these guarantees may seem simple, providing them while maintaining performance is non-trivial. We present three different methods of providing reliable service within the MPI library.

**Thread**: This is the traditional approach in which a watchdog thread sleeps for a set amount of time before waking to process incoming messages, send ACKs, and resend timed-out messages. This method provides a solution to both of the issues of sender and receiver progress, but it does so at significant cost. To maintain a quick response to a segment drop the thread wakeup time, $t_{wake}$, must be set to a short interval. A thread that wakes up frequently may harm performance. Figure 5.3(a) demonstrates this protocol.

**Progress-Engine**: The progress engine of MPI refers to the core of the library that is activated to send and receive messages – to make progress on communication. In this




(a) Thread Based (b) Progress Engine Based (c) Hybrid Event Based

Figure 5.3: Reliability Protocols

variant, reliability is only handled in the progress engine. This has a number of advantages including the lack of the overhead of locks for various data structures that otherwise need protection. Additionally, for infrequent packet drops, there is no need to have a thread wake up frequently, which causes potential performance degradation. Since we are relaxing the timeframes for retransmission and acknowledgment, the disadvantage of such a design is that an ACK may not be returned within the timeout period since the receiver may be in a computation loop (not discovering the message), triggering an unnecessary retransmission from the sender. Additionally, sender retransmission may also be delayed if the sender is in a long compute loop. Figure 5.3(b) shows this protocol when the receiver is in a long compute loop and an unnecessary retransmission is generated.

**Hybrid**: In this method the reliability is primarily through the progress engine; however, if the timer runs out the message is retransmitted to the receiver side to a secondary UD QP that is interrupt-driven. When a message is received on the interrupt-driven QP a thread is awoken to process the message and acknowledge all other waiting messages. This guarantees an ACK will be sent back within a Round Trip Time (RTT) if received,

reducing unnecessary multiple retries while avoiding the overhead of a thread waking at a constant rate. Figure 5.3(b) shows the benefit of this method when the receiver is in a compute loop, avoiding multiple retransmissions. Using an interrupt-driven QP for all communication is not feasible since interrupts are costly operations that harm performance of both communication and computation.

Note that implementations of both the *thread* and *hybrid* designs require locking of internal data structures to ensure thread safety. Using the *progress engine* method avoids this as only one thread is only ever executing within the progress engine.

## 5.3 Implementation

We have implemented our proposed design over the verbs interface of the OpenFabrics/Gen2 stack [55]. Our prototype design is based on MPICH [24] from Argonne National Laboratory. MPICH defines an Abstract Device Interface (ADI), that allows the device and transport specific information to be encapsulated. We develop an ADI component based on the MVICH [34] and MVAPICH ADI components. MVICH was developed at Lawrence Berkeley National Laboratory for the VIA [18] interconnect. MVAPICH is a derivative of MVICH from the Ohio State University that has been significantly redesigned and optimized for InfiniBand. Both MVICH and MVAPICH were created for reliable connection-oriented transports, requiring a significant portion of the code path to be re-written to support UD.

We implement all three of the reliability methods described in the design section as well as a profiling engine to detect message retransmissions, duplicate messages, and message sizes. We compute message drop rates by comparing the number of re-transmissions from

a given process to another and the number of duplicate messages the process detected from the other. This data is collected during the finalize stage.

## 5.4 Evaluation

Our experimental testbed is an 1152-node, 9216-core, InfiniBand Linux cluster at Lawrence Livermore National Laboratory (LLNL). Each compute node has four 2.5 GHz Opteron 8216 dual-core processors for a total of 8 cores. Total memory per node is 16GB. Each node has a Mellanox MT25208 dual-port Memfree HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55]. The Intel v9.1 compiler is used for all compilations, including MPI libraries and applications.

We compare the performance of our UD-based prototype versus the popular MVAPICH MPI library, version 0.9.8. Comparisons with other MPIs were not possible due to limited time on the cluster. For MVAPICH, we disable small message RDMA, since per process it requires roughly 512KB of memory per additional connected process; large message RDMA for zero-copy transfers is still enabled. We disable this feature since we are interested in ultra-scale clusters. Earlier studies [78] have shown the SRQ path to have similar or better performance for applications than the small message RDMA path while using significantly less memory. Small message RDMA, with lower latency (3.0$\mu$sec for RDMA, 5.0$\mu$sec for SRQ), may be beneficial for smaller clusters, but is not scalable for large-scale systems due to the large per process memory usage.

Additionally, although by default MVAPICH posts 512 buffers to the shared SRQ for receiving messages, we had to increase that value to 8K to ensure optimal performance for some applications on this larger cluster.

Table 5.1: Additional Memory Required with additional processes and connections

| | Memory Required Per Process | |
|---|---|---|
| MPI | Addl. Peer | Addl. Connection |
| MVAPICH-0.9.8 | 0.61KB | 68KB |
| MVAPICH-opt | 0.61KB | 13KB |
| UD | 0.44KB | 0KB |

### Memory Usage

One of the primary benefits of using UD as the transport protocol is the reduced memory usage, as noted in Section 5.1.1. In this section we describe our methodology and compare the memory usage of the entire MPI library at increasing scale.

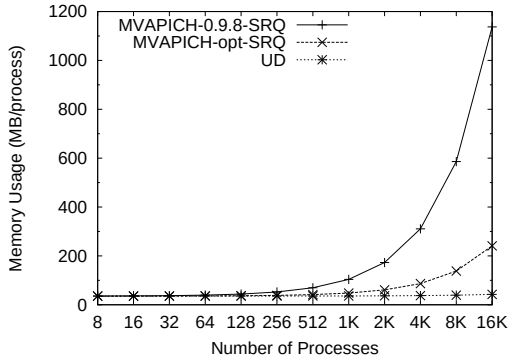


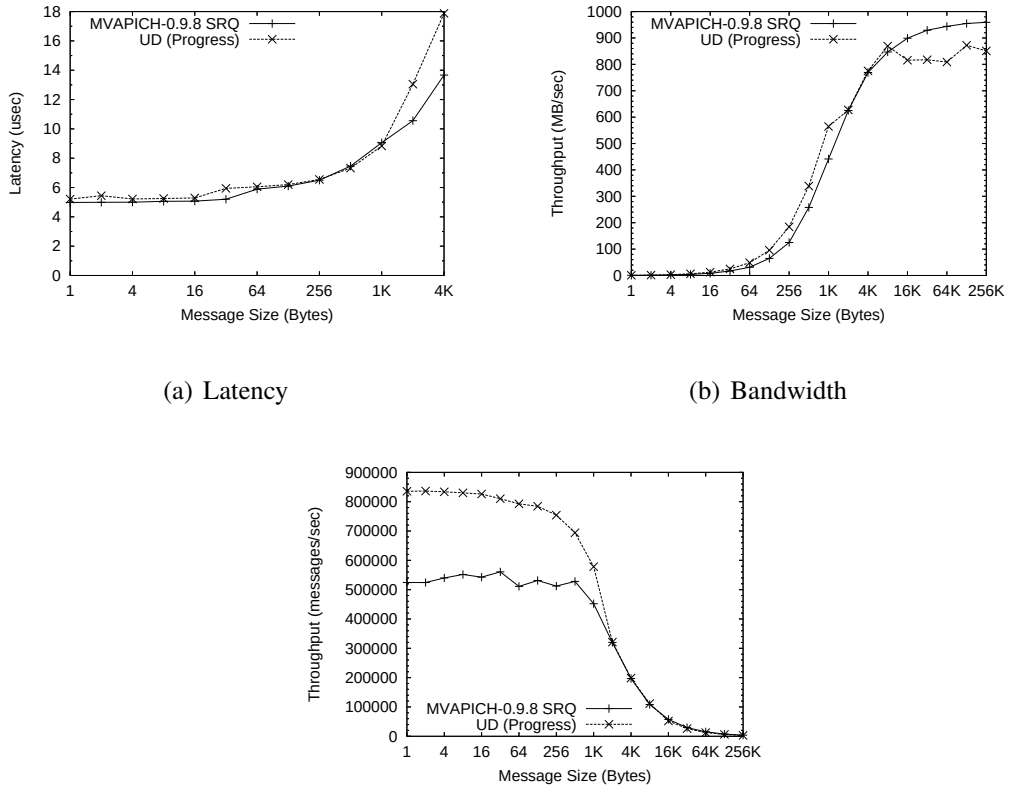

Figure 5.4: Fully-Connected Memory Usage

Figure 5.4 compares the memory usage of our proposed design as compared to MVAPICH with increasing numbers of connected peers. Values through 4K processes are measured, 8K and 16K values are modeled. From the figure we observe that MVAPICH has

the potential to use 1.1GB of memory per process at 16K processes per job, and our prototype uses only 40MB per process. Expanding to total connection memory usage across the cluster, which is what scientists use to determine the problem size able to be run, the default settings will consume 18TB of memory, and the UD-based design will use a much smaller 655GB of memory. This memory usage is the worst-case situation, where all peers are communicated with at least once. As mentioned in Section 5.1.1, many applications communicate with the majority of their peer processes, so the resource usage is important to consider and cannot be ignored.

Settings currently being used on our evaluation cluster for reduced memory usage are shown as "MVAPICH-opt", these decrease the number of outstanding send operations allowed per QP as well as the amount of data that can be inlined with a request to decrease overall memory usage significantly from the 0.9.8 default. This decreases the total connection memory, however, we have not evaluated all possible effects. Our evaluation has been done with the default settings; we mention possible optimization here for completeness. Even using these optimized settings our UD-based design uses 80% less memory at 16K processes. A subsequent release of MVAPICH (0.9.9) has reduced memory usage below even the level of the optimized settings, however, UD remains the most scalable option.

We also calculate the memory usage of MVAPICH when the number of processes in a job increases, but no direct connections are made to any peers. This reflects only the *data structure* memory. Table 5.4 shows the results of this measurement. From this table we observe that our UD-based prototype uses even less memory in data structures than MVAPICH, meaning even with no connections setup the memory of our UD prototype uses less memory than MVAPICH as the number of processes in a job increases.




(a) Latency

(b) Bandwidth

(c) Messaging Rate

Figure 5.5: Basic Micro-Benchmark Performance Comparison

### 5.4.1 Basic Performance

Figure 5.5 shows the latency, bandwidth, and messaging rate of both MVAPICH (with SRQ) and our UD-based prototype using the *progress engine* based reliability method. From Figure 5.5(a) we observe that for small messages under the MTU size message latency is nearly the same as MVAPICH. Messages above that size have a higher latency as the library must segment the message instead of segmenting in hardware. We also evaluate the latency penalty of using locks around all critical data structures for the *thread* and *hybrid*

reliability methods and find it to have a 0.2 $\mu$sec additional overhead. Figure 5.5(b) shows the bandwidth comparison. For messages less than the MTU size our prototype shows increased throughput over MVAPICH. Throughput is roughly comparable until 8KB where MVAPICH maintains a bandwidth of 100-150 MB/sec higher. From Figure 5.5(c) we observe that the messaging rate of our prototype is up to 40% higher for small messages. After studying the base InfiniBand performance with and without a SRQ, it can be observed that the lesser small message bandwidth and message rate for MVAPICH is due to the use of a SRQ. Recall from Section 2.1.3 that for scalability a SRQ is required when using RC.

### 5.4.2 Application Results

We evaluate our designs with the NAS Parallel Benchmarks, and three applications from the Advanced Simulation and Computing (ASC) Benchmark Suite [4]: sPPM, Sweep3D, and SMG2000. Given limited large-scale cluster time, sPPM, Sweep3D, and SMG2000 are evaluated with only the *progress-engine* approach for reliability and MVAPICH. For the thread-based reliability method we set the thread-wake time to one second and a message timeout of two seconds. The progress engine and hybrid message timeout is set to 0.4 sec. All results are the average of multiple runs.

For each of the applications we evaluate the memory usage in all three categories: data structure, connection/context memory and communication buffer memory used by the MPI library. Additionally, we profile the MPI message sizes used by the application and directly communicating peers to better understand the results. Communication pattern analysis of sPPM, SMG200, and Sweep3D is available in [84] by Vetter, et al.

Table 5.2: NAS Characteristics (256 processes)

| Characteristic | Benchmark | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | BT | CG | EP | FT | IS | LU | MG | SP |
| Total MPI Messages (millions) | 4.96 | 13.31 | 0.01 | 1.47 | 1.50 | 77.31 | 1.68 | 9.88 |
| RC: Connections (max per process) | 12 | 9 | 8 | 255 | 255 | 10 | 13 | 12 |
| UD: Total Packet Drops | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**NAS Parallel Benchmarks**

The NAS Parallel Benchmarks [8] (NPB) are kernels designed to be typical of various Computational Fluid Dynamics (CFD) MPI applications. As such, they are a good tool to evaluate the performance of the MPI library and parallel machines. We use the largest class (C) with datasets for all benchmarks. Table 5.2 shows the number of messages, RC connections, and UD packet drops observed for 256 processes. Additional study on communication patterns of NAS Parallel Benchmarks is available in [85].

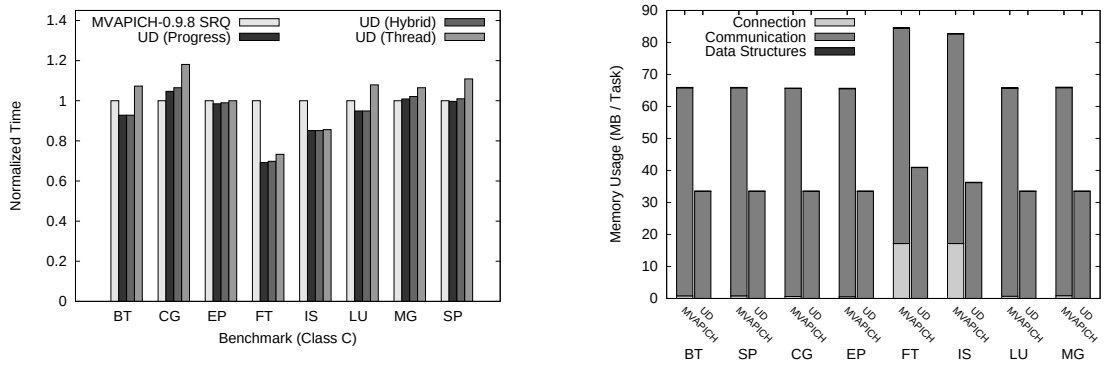
We evaluate the performance of each of the benchmarks with both our UD prototype and MVAPICH for 256 processes on 256 nodes. Each of the design alternatives for reliability are evaluated for the UD prototype. Figure 5.6(a) shows the results of our evaluation. Comparing only the reliability methods for the UD prototype, the *progress-engine* approach gives the best observed performance for all benchmarks. The other techniques require additional locking overhead for thread safety. The thread-based technique has the highest overhead, sometimes reaching nearly 10%. We attribute this to additional system noise [58] incurred by the threads waking up and competing for computational cycles and locks. The hybrid approach incurs very little overhead as no retransmissions are made.

Comparing the UD prototype performance with MVAPICH we observe that our UD prototype performs quite well, performing with better or equal performance in all benchmarks besides CG and MG where performance is within 3%. The UD prototype performs particularly well for FT and IS, both of which make use of `MPI_Alltoall` operations, where an RC-based MPI will have to go outside the ICM cache to communicate with all other nodes.

Figure 5.6(b) shows the memory usage of each of the benchmarks. As noted earlier, we measure the connection memory (for QPs), the communication buffers, and the memory required for data structures. We observe that even for MVAPICH, the connection memory is not significant for many of the benchmarks since each process has only a few communicating peers. IS and FT, however, connect to all 255 other peers, making their connection memory more significant when the RC transport is used. The UD-based prototype shows negligible connection memory in all cases since it is based on a connection-less transport. The communication buffer memory usage is higher for MVAPICH as each receive buffer posted is 8KB and those for our UD prototype are 2KB (the MTU size). Some applications that use a significant number of large messages, such as FT and IS, required additional memory above the base value for the UD prototype. Overall, including all forms of communication memory the UD prototype used lower memory for all benchmarks.

**sPPM**

sPPM [49] is an application distributed as part of the ASC Purple Benchmarks. It solves a 3D gas dynamics problem on a uniform Cartesian mesh using the Piecewise Parabolic Method (PPM). Figure 5.7 shows a breakdown of MPI message volume according to size for this dataset at 4K processes. We observe from the figure that 40% of messages are 64 bytes or less and another 40% are larger than 256 KB.




(a) Normalized Time

(b) Memory Usage Per Process

Figure 5.6: Characteristics of NAS Benchmarks (256 processes, Class C)

The performance of sPPM with increasing numbers of processes is shown in Figure 5.8. From the figure we observe that performance between the default configuration of MVAPICH and our prototype is similar. For 4K processes, the UD prototype is within 1% of the performance of MVAPICH. The lower performance and lack of zero-copy for large message transfers with UD and the high percentage of messages over 256KB likely combine for the observed lower performance. Table 5.3 shows statistics related to the MPI messaging characteristics of sPPM. Memory usage per process is nearly constant, regardless of the total number of processes. Given the low number of communicating peers for each process, maximum 15, the MVAPICH RC connection memory is kept less than 1MB. Communication buffer memory per process is 68MB for MVAPICH and 36MB for our UD prototype. No packet drops were observed for any number of processes.

Table 5.3: Application Characteristics

| Application | Characteristic | Processes | | | | | |
|---|---|---|---|---|---|---|---|
| | | 128 | 256 | 512 | 1024 | 2048 | 4096 |
| sPPM | Total MPI Messages (millions) | 0.11 | 0.24 | 0.52 | 1.08 | 2.25 | 4.74 |
| | RC: Connections (max per process) | 10 | 10 | 12 | 13 | 14 | 15 |
| | UD: Total Packet Drops | 0 | 0 | 0 | 0 | 0 | 0 |
| Sweep3D | Total MPI Messages (millions) | 0.17 | 0.32 | 0.63 | 1.28 | 2.65 | 5.48 |
| | RC: Connections (max per process) | 9 | 9 | 9 | 9 | 10 | 11 |
| | UD: Total Packet Drops | 0 | 0 | 0 | 0 | 1 | 8 |
| SMG2000 | Total MPI Messages (millions) | 29.9 | 64.4 | 146.1 | 312.7 | 652.1 | 1376.6 |
| | RC: Connections (max per process) | 111 | 195 | 340 | 438 | 631 | 992 |
| | UD: Total Packet Drops | 0 | 0 | 0 | 0 | 10 | 27 |

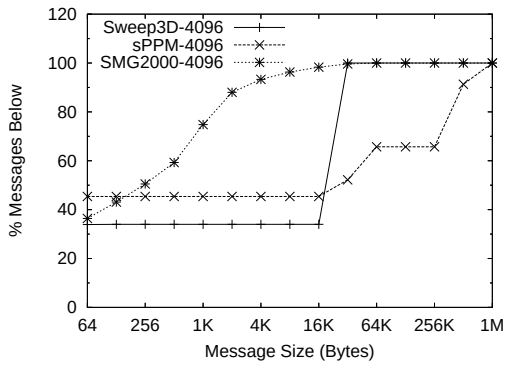



Figure 5.7: Total MPI Message Size Distribution

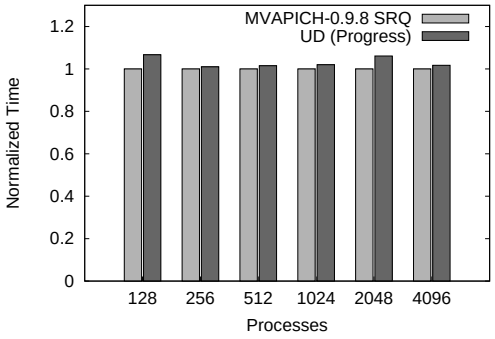



Figure 5.8: sPPM Performance Comparison

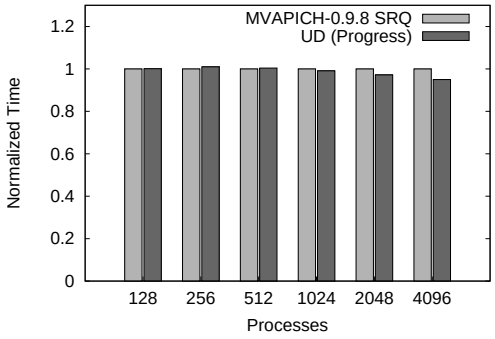
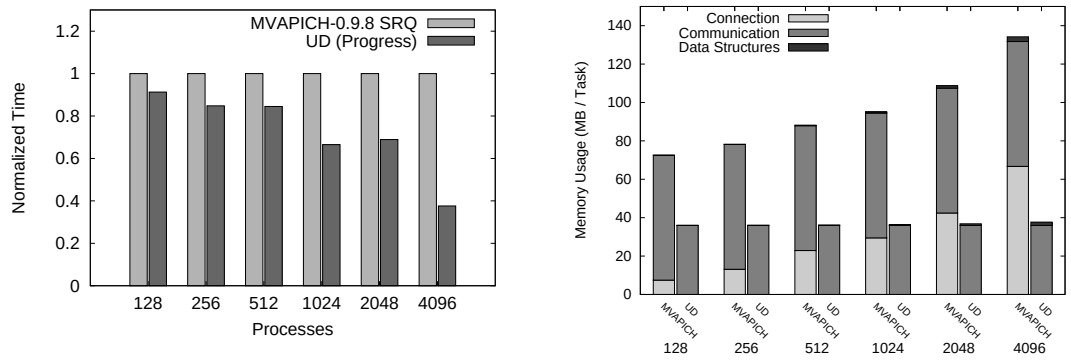



Figure 5.9: Sweep3D Performance Comparison

### Sweep3D

Sweep3D [25, 32] uses a multi-dimensional wavefront algorithm to solve the 3D, time-independent, particle transport equation on an orthogonal mesh. Profiling of the MPI message sizes, shown in Figure 5.7 reveals that 38% of messages are 64 bytes or less and the remaining volume is between 32 and 64 KB in size.

Figure 5.9 shows the normalized time of running Sweep3D at increasing scale. Performance is roughly comparable, with the UD-based design getting slightly faster with increasing numbers of processes: 5% faster at 4K processes. From Table 5.3 we observe that the as the number of processes in a job increases, the number of packet drops increases. The percentage is still low, at 4K processes only 8 packets were dropped. As with sPPM, the memory required for RC connections for MVAPICH is quite low with a maximum of 11. As such, the difference in memory required for the UD prototype and MVAPICH is minimal.




(a) Performance Comparison

(b) Memory Usage

Figure 5.10: SMG2000 Characteristics with increasing processes

**SMG2000**

SMG2000 [15] is a parallel semi-coarsening multigrid solver, which is part of the ASC Purple Benchmarks. Analysis of the MPI message distribution, as shown in Figure 5.7 reveals the majority of messages are quite small, 90% of MPI messages are less than 4KB.

From Figure 5.10(a) we observe superior performance for our UD-based design as the number of processes increases. At 4K processes our prototype is 60% faster than MVAPICH. As noted in Table 5.3, the number of connected peers is significant, nearly 1000 connected peers per process for 4K processes. Further inspection of the communication pattern shows a regular frequency of communication with many peers. As such, the QP context caching and management of ACKs at the application layer is the likely reason for the benefit, particularly since the size of each message is very small. Unfortunately, there is no direct mechanism to measure the cache misses on the HCA ICM cache. When sending to such a large number of peers the ICM cache is likely being thrashed for RC, while the

UD QPs will remain in cache. We also observe increasing numbers of packet drops with scale; at 2K processes 10 packets are dropped and 27 packets are dropped for 4K processes.

Figure 5.10(b) shows the memory usage of SMG2000 with increasing numbers of processes. Since the number of connected peers increases with the overall number of processes, the connection memory similarly increases for MVAPICH. The UD prototype maintains constant connection memory. At 4K processes, the maximum number of RC connections for MVAPICH made by a single process is nearly 1000, requiring over 60MB of memory per process just for connections, totaling 240 GB of memory across the cluster.

## 5.5 Zero-Copy Design

As noted in Section 2.1.2, the UD transport does not allow RDMA semantics. Also, InfiniBand/OpenFabrics [55] does not implement any Matched Queues interface. Thus, two of the popular methods mentioned in Section 2.3 to design a zero-copy protocol are not available over UD transport. However, for scalability reasons mentioned in the previous sections, using the UD transport is highly desired on very large scale clusters. In this section we present a novel design for zero-copy protocol over UD transport. We first describe the various design challenges in detail, followed by the proposed design.

### 5.5.1 Design Challenges

The challenges of providing high-performance reliable transport at the software level over UD are significant. The first UD-based design, presented in Section 5.2 used the traditional TCP-style reliability over UD with significantly lower performance for large messages due to extra memory copies. In this section we discuss these challenges that must be overcome for a high-performance true zero-copy design.

**Limited MTU Size**: A significant disadvantage of using the UD transport is that messages are limited to the maximum MTU size. The maximum MTU available on current Mellanox HCAs is 2 KB; the specification allows for an MTU of up to 4 KB. Segmentation of larger messages is required before posting the send operation since the hardware will not perform this action for UD.

**Lack of Dedicated Receive Buffers**: As discussed in Section 2.1.1, to receive messages using channel semantics the receive buffers must be posted to a QP. While using the UD transport, the UD QP is shared for all remote peer processes. In this scenario, it is difficult to post receive buffers for a particular peer as they are all shared. Additionally, using the UD transport, if no buffer is posted to a QP, any message sent to that QP is silently dropped. A high-performance design must always strive to avoid dropping packets due to unavailability of buffers.

**Lack of Reliability**: There are no guarantees that a message sent will arrive at the receiver. Traditional approaches used by TCP-style reliability algorithms can assist in achieving reliability, but adding packet headers for sequence numbers, acknowledgments, and tracking send operations is non-optimal for high-performance. Adding these fields inside packets can lead to memory copies, since these fields would have to be placed in each packet individually.

**Lack of Ordering**: Messages may not arrive in the same order they were sent. Each buffer is consumed in a FIFO manner, so the first buffer filled may not be the first packet sent from a process. The MPI library must now take care of re-arranging the arrived packets in the correct order and form the message from the individual packets.

**Lack of RDMA**: The lack of RDMA is the most significant deficiency of UD in terms of implementing a high-performance large message transfer mechanism. If RDMA were provided the sender and receive buffers could be pinned and transferred directly without regard to ordering since messages are directly placed in the user buffer. Reliability could be ensured through tracking of completions on the receiver side. This mechanism is similar to the implementation of RDMA on the IBM HPS adapter where the HCA can directly place the incoming data to the user buffer since each packet describes its own placement location [73]. Unfortunately, InfiniBand does not provide such a capability for UD.

**Extra headers in application UD packets**: The Global Routing Header (GRH) is used by InfiniBand switches to route packets between subnets. The InfiniBand specification mandates that these headers be present in UD based multi-cast packets. However, current InfiniBand hardware and software stack place this 40-byte GRH header inside the application packets. This is particularly frustrating since even if reliability and ordering were guaranteed, it would not be straight-forward to simply post the user application receive buffers directly to the QP.

### 5.5.2 Proposed Design

Large message transfers are typically done through a "rendezvous" protocol: the sender notifies the receiver of intent to send a message, a Request To Send (RTS). The receiver will reply with a Clear To Send (CTS) message as soon as the receive has been posted. Such a technique is used in traditional large message transfers in InfiniBand to perform an RDMA Write/Read operation to directly place the data into the receive buffer. In this case the CTS message will contain the address of the receive buffer to allow RDMA operations. Finally, a finish message (FIN) is sent to notify the receiver of data placement.

Our design uses a similar CTS/RTS protocol as is common in other MPI implementations, however, there are important differences due to the lack of RDMA and unreliable transfer. An overview of our proposed protocol is shown in Figure 5.11.

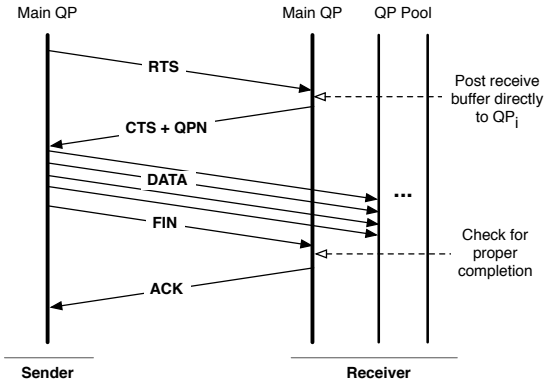



Figure 5.11: UD Zero-Copy Protocol

### Enabling Zero-Copy

Our proposed design to implement zero-copy transfers over UD is based on a *serialized communication* model since RDMA and tag matching are not specified for the UD transport. "Serialized" is used here to denote that communication on a given QP can be serialized – the order of transfer is agreed on beforehand and only one sender will transmit to a QP at a single time.

After receiving a RTS message from a sender, the receiver will select a UD QP for this transfer; we will denote this $QP_{transfer}$. If the receive operation as not been posted, the receiver queues the RTS message until a matching receive is posted. When the receive is posted, the buffer is directly posted in MTU size chunks to $QP_{transfer}$. In this way, the first message sent to $QP_{transfer}$ will be directly placed into the first MTU size chunk of the

receive buffer. The rest of the received data will continue to fill the buffer in the order of arrival.

**Efficient Segmentation**

One of the challenges for using the UD transport is the lack of support for messages larger than the MTU size. Although this is not a problem that can be solved without changing the specification, steps can be made to reduce the detrimental effects. One of the major overheads is the mechanism used to report completion of packet send operations. Since large messages can generate lots of packets, the overhead might be higher. In our design, we choose to get "completion signal" only for the last packet of the message. The underlying reliability layer would mark packets as missing at the receiver side and the sender would be notified. Hence, the intermediate completions at the sender are unnecessary, as long as we know the last packet was sent out of the node, it is enough.

**Zero-Copy QP Pool**

Receive requests for UD QPs are shared for all remote peers. This makes it harder to dedicate a set of resources for a particular remote process for sending large messages. In order to solve this problem, we maintain a pool of UD QPs which can be used as required. When a large message transfer is initiated, a QP is taken from the pool and the application receive buffer is posted to it (in MTU chunks). For the duration of the message transfer, this QP is bound to the specific remote process. When the transfer is finished, the QP is returned to the pool. Obtaining and returning QPs from a pool are very cheap, hence our design is expected to perform reasonably well for up to as many large incoming sends as the size of the pool. For the purpose of our evaluation, the pool size is 64, although this is tunable at runtime.

### Optimized Reliability and Ordering for Large Messages

As mentioned earlier in this section, the UD transport does not guarantee any reliability or ordering. In our zero-copy protocol, we need to take care of these issues for maintaining data integrity and MPI semantics.

The first challenge is determining if message re-ordering has occurred. One method would be to perform a checksum of the final receive buffer, however, such a scheme negates much of the benefit of using a zero-copy protocol since the data buffer must now be traversed. Instead we propose leveraging the *immediate data* field of InfiniBand. In addition to the normal data payload, send operations can also specify a 32-bit immediate field that will be available to the receiver as part of the completion entry. In our design the immediate field is filled with a sequence number that identifies the order in which data was sent and should be received. Since each QP is associated with a CQ, upon receiving the FIN message the associated CQ can be polled for completions. If any completion is out of order or missing, re-ordering or drops have occurred during transfer. If a message has been dropped or re-ordered there are various techniques that can be used. The most simple, and the approach we have implemented, will send a negative acknowledgment (NACK) to the sender to request retransmission of the entire message. More optimized approaches, such as copying only re-ordered parts of the receive buffer or selective retransmission may be helpful in high-loss environments. In our previous work, however, we have observed extremely low data loss for even large-scale systems with thousands of processes. Upon a successful reception of the message, the receiver will send an ACK to the sender, allowing the sender to mark the send operation complete.

#### Dealing with GRH Data

As noted in Subsection 5.5.1, the GRH data is placed into the first 40 bytes of the receive buffer. Since the GRH data is for routing and not data that we wish to have in the receive buffer, we must avoid placing the GRH data there. As such, we use a *scatter list* of InfiniBand to specify two buffers in a receive descriptor – the first with a length of 40 bytes that maps to a temporary buffer and a second entry that maps to the receive buffer.

## 5.6 Zero-Copy Evaluation

Our experimental test bed is an InfiniBand Linux cluster. Each compute node has dual 2.8 GHz Opteron 254 single-core processors. Each node has a Mellanox MT25208 dual-port Memfree HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55], OFED 1.1 release. The proposed design is integrated into the UD communication device of MVAPICH [53] previously designed in Section 5.2.

We evaluate the following three transfer methods at the MPI layer:

- *Reliable Connection (RC-rdma)*: For measuring RC MPI characteristics we use MVAPICH 0.9.9. Large messages are transferred using the rendezvous RDMA write operation described in Section 2.3.

- *Unreliable Datagram - Copy-Based (UD-copy)*: The MPI library used for evaluating this is the same as used in Section 5.2, which was shown to scale to thousands of processes with good performance.

- *Unreliable Datagram - Zero-Copy (UD-zcopy)*: This is an implementation of the proposed design in Section 5.5.2. Aside from the zero-copy protocol, the MPI library

is identical to that of UD-copy. The zero-copy threshold is set to be 32 KB, meaning all messages 32 KB and above will use the zero-copy protocol.

### 5.6.1 Basic Performance

In this subsection we evaluate the microbenchmark performance of latency, uni-directional bandwidth, and bi-directional bandwidth using the OSU Benchmarks [54].

#### Latency

To measure latency we use a ping-pong test and report one half of the value as the one-way latency. The latency results for large messages are shown in Figure 5.12. Latencies for messages smaller than 32 KB do not change with our method since we have set the threshold to 32 KB for zero-copy transfers. The gap between UD-copy and RC-rdma is very significant – UD-copy latency is nearly double that of RC-rdma for a 1 MB message. Due to the overhead incurred through segmentation and posting of receive buffers, UD-zcopy does not fully close the gap. The proposed design, however, shows only a 28% increase in latency over the RC-rdma implementation, a significant improvement over the copy-based approach.

When compared to RC-rdma, the performance of the UD-zcopy design performs very well, although many overheads discussed in Section 5.5.2 do reduce the performance from the RC-rdma level. In particular, posting explicit receive buffers instead of RDMA and an additional DMA operation to split the GRH field are significant costs.

#### Uni-Directional Bandwidth

In the uni-directional bandwidth benchmark, we evaluate the bandwidth of increasing message sizes for each of the transfer methods. This benchmark performs a throughput

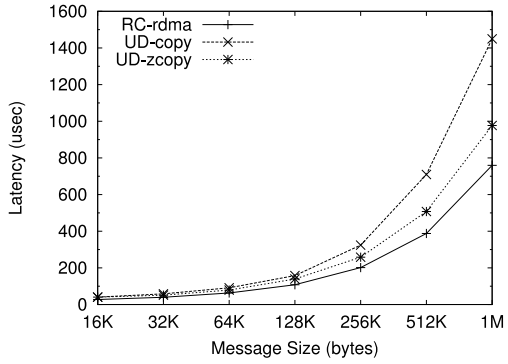



Figure 5.12: One-way Latency

test between a sender and receiver where for each message size the sender sends a window of 64 messages and waits for an acknowledgment of the entire window from the receiver. This step is repeated for many iterations and the average is reported.

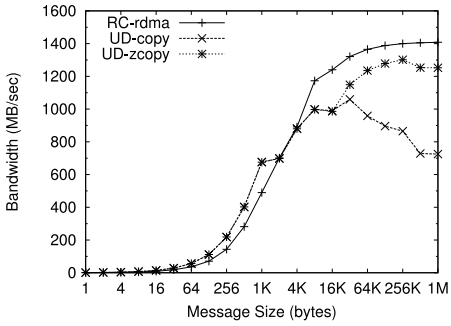



Figure 5.13: Uni-Directional Bandwidth

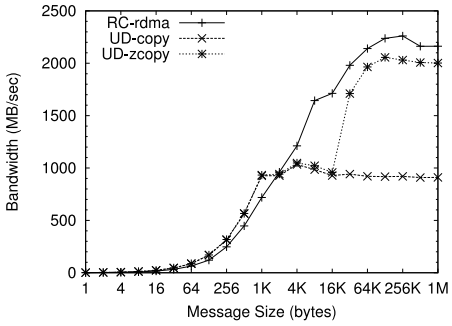



Figure 5.14: Bi-Directional Bandwidth

Figure 5.13 shows the results of the experiment. We can clearly observe the 32 KB threshold where the zero-copy protocol begins to be used. Already at 64 KB the difference between the UD-copy and UD-zcopy methods reaches 275 MB/sec, a nearly 30% improvement. The difference is even greater for larger message sizes; 1 MB messages are transferred over 500 MB/sec faster using the proposed UD-zcopy design. Due to the various overheads incurred, the bandwidth as compared to RC-rdma is lower, however, the performance is within 9% of the performance at 64KB.

**Bi-Directional Bandwidth**

The bi-directional bandwidth benchmark is similar in design to the uni-directional bandwidth benchmark presented earlier. However, in this case, both sides of the communication send as well as receive data from the remote peer.

From Figure 5.14 we can observe a steep increase in performance at the zero-copy threshold of 32 KB. At 64 KB the difference in performance between UD-copy and UD-zcopy is over 1 GB/sec. As compared to RC-rdma, the UD-zcopy design is slightly worse – as expected from the additional overheads, however, for all message sizes where the zero-copy protocol is being used the bandwidth difference is only 150 MB/sec, or a 7% degradation for a 1 MB message versus a nearly 60% degredation for the copy-based approach.

### 5.6.2 Application Benchmarks

In this section we evaluate the three messaging transfer modes against application-based benchmarks. These are more likely to model real-world use than microbenchmarks. We evaluate a molecular dynamics application and the NAS Parallel Benchmarks (NPB).

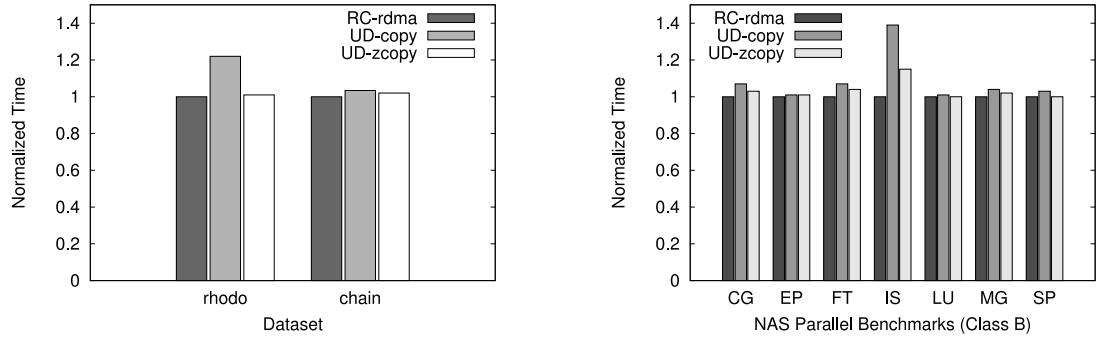
Table 5.4: Messaging Characteristics (16 processes)

| App. | Benchmark | Percentage | | | Total Messages | | |
|---|---|---|---|---|---|---|---|
| | | 0-2KB | 2-32KB | 32KB+ | 0-2KB | 2-32KB | 32KB+ |
| LAMMPS | in.chain | 29.65 | 70.35 | 0.00 | 166308 | 394618 | 0 |
| | in.rhodo | 41.37 | 47.31 | 11.31 | 150238 | 171804 | 41088 |
| NPB | CG.B | 57.62 | 0.00 | 42.38 | 257902 | 0 | 189696 |
| | EP.B | 100.00 | 0.00 | 0.00 | 384 | 0 | 0 |
| | FT.B | 19.50 | 0.00 | 80.50 | 1364 | 0 | 5632 |
| | IS.B | 61.77 | 6.04 | 32.20 | 5402 | 528 | 2816 |
| | LU.B | 99.00 | 0.00 | 1.00 | 2401046 | 0 | 24192 |
| | MG.B | 57.99 | 24.74 | 17.28 | 56706 | 24192 | 16896 |
| | SP.B | 0.48 | 0.00 | 99.52 | 744 | 0 | 154080 |

**LAMMPS Benchmark**

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [60] is a classical molecular dynamics simulator from Sandia National Laboratories. Several benchmark datasets are available from the LAMMPS website. They are meant to represent a range of simulation styles and computational expense for molecular-level interaction forces. In our experimental analysis, we used the Rhodospin protein (`in.rhodo`) and Polymer chain melt (`in.chain`) datasets available from the LAMMPS website. LAMMPS reports the "Loop Time" for a particular benchmark as a measure of CPU time required to simulate a set of interactions.

Figure 5.15(a) shows the normalized time to complete the simulation and Table 5.4 shows the messaging characteristics of the different data sets. For the `in.rhodo` dataset the performance of UD-zcopy is significantly increased over that of UD-copy and near equal to that of RC-rdma. This improvement can be attributed to the 11% of messages that are greater than or equal to 32 KB and are benefiting from the zero-copy transfer mechanism.




(a) LAMMPS

(b) NAS Parallel Benchmarks

Figure 5.15: Evaluation Normalized Time (16 processes)

Since the `in.chain` dataset does not have messages 32 KB or over, the performance of both UD-copy and UD-copy are the same. RC-rdma performs similarly due to the lack of large messages as well.

**NAS Parallel Benchmarks**

The NAS Parallel Benchmarks [8] are a selection of kernels that are typical in various Computational Fluid Dynamics (CFD) applications. As such, they are a good tool to evaluate the performance of the MPI library and parallel machines.

The results of evaluating each of the methods on the NAS benchmarks can be found in Figure 5.15(b). For many of the benchmarks where there is a significant number of large send operations (FT, IS, and CG), we see a large difference in performance between UD-copy and RC-rdma – for IS the performance of UD-copy is nearly 40% worse. Our proposed zero-copy design, however, reduces this overhead considerably. Performance is

increased by 4%, 3%, 16%, 2%, and 3% for CG, FT, IS, MG, and SP, respectively. As we can observe from comparing Table 5.4 and our percentage improvement, although some benchmarks use many large messages, such as SP, it may not be bandwidth limited and communication can be overlapped with communication.

Aside from IS, where the application is extremely short running and the communication time can become dominated by setup overheads, the absolute application time is very similar between RC-rdma and UD with the addition of the zero-copy protocol.

## 5.7 Related Work

The issue of memory consumption for connection-oriented transports has also been studied by other researchers. Gilfeather and Maccabe proposed a connection-less TCP method that activates and deactivates connections as needed to save memory [22]. Scalability limitations of the connection-oriented VIA interconnect were explored in [12] and an on-demand connection management method for VIA was proposed in [89].

InfiniBand, the successor of VIA, has also been studied in regards to resource usage. Early versions of MVAPICH exhibited significant memory usage as the number of connections increased as studied by Liu, et al in [36]. Yu, et al. proposed an adaptive connection setup method where UD was used for the first sixteen messages before an RC connection was setup [90]; in this case RC was the primary transport. Work by Sur, et al. in [75] significantly reduced the per connection memory usage in MVAPICH using InfiniBand Shared Receive Queue (SRQ) support and a unique flow control method. Similar techniques have been used in Open MPI by Shipman, et al in [69]. In both of these studies, the memory usage was reduced mostly through a reduction in communication memory buffer usage while

still using the RC transport layer. We instead focus on reducing the connection memory usage by leveraging the connection-less UD transport.

Other researchers have explored providing reliability at the MPI layer as part of the LA-MPI project [23, 6], however, their work had a different goal. LA-MPI is focused on providing network fault-tolerance at the host to catch potential communication errors, including network and I/O bus errors. Their reliability method works on a watchdog timer with a several second timeout, focusing on providing checksum acknowledgment support in the MPI library. Conversely, in this work we explore using the connection-less UD transport of InfiniBand to provide superior scalability and near-equal to better performance compared to RC for ultra-scale clusters.

UD has also been used in other work to suport enhanced collective operations for MPI. Researchers have previously shown the benefit of using hardware-based multicast over UD to increase collective performance in MVAPICH [38, 44]. Mamidala, et al., has also shown the benefit of using UD for `MPI_Alltoall` operations [45]. Our work instead focuses on a complete MPI stack over UD for high-performance and scalable point-to-point performance.

Zero-copy protocols have been widely used in MPI implementations. MPICH-PM [16] used a zero-copy protocol over the PM communications library. The design of the protocol uses a Request to Send (RTS), Clear to Send (CTS) and Send FIN messages. MPI-BIP [61] also used zero-copy message transfers to reduce copy overhead and boost application performance. This work noted that the communication bandwidth was comparable to memory bandwidth and thus, zero-copy would be an attractive protocol. This observation holds good for modern systems. RDMA had been emulated over InfiniBand UD in [45] using a copy-based method, however no zero-copy operations were designed or evaluated. RDMA

is provided using a UD-based approach with reliability provided by MPI in the IBM HPS adapter [73], however, each UD packet is self-identifying as to its address. This is unlike InfiniBand where RDMA is not enabled for UD.

Although these research works focused on utilizing zero-copy protocols for performance reasons, there was no special treatment of these protocols with regard to scalability. While there are many research works which focus individually on zero-copy protocols and scalability of MPI libraries, to the best of our knowledge, this is the first work on combining both these objectives in a scalable, high-performance MPI design and implementation for InfiniBand.

# CHAPTER 6

# UNRELIABLE CONNECTION: HARDWARE VS. SOFTWARE RELIABILITY

Given that both the MPI library and the network interconnect are so critical to performance in cluster environments, it is important to examine the assumptions traditionally made in each. In many modern interconnects, such as InfiniBand and Quadrics, the most commonly used interface is a reliable one where reliability is provided in the network device hardware or firmware. With this hardware support the MPI can be designed without explicit reliability checks or protocols.

In this chapter we examine this common assumption. InfiniBand hardware supports three transports, Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). On Mellanox [1] hardware, the reliability for the RC transport is handled in firmware on the network device. Given this, we use InfiniBand as a platform to evaluate the tradeoffs associated with providing reliability in the MPI layer as compared with a hardware-based design. We compare all three InfiniBand transports, RC, UC and UD in a common codebase with and without software reliability on a variety of communication patterns in this study. To the best of our knowledge, this is the first work to address this issue in such a comprehensive manner on high-speed interconnects.

There are two main questions this work seeks to address:

- Can software-based reliability outperform hardware-based reliability methods, and if so, under what conditions?
- What are the costs associated with providing reliability in software?

As part of our work we design an MPI library that allows independent evaluation of each of the three InfiniBand transports in a common codebase. Additionally, software-level reliability can be turned on and off for each of the transports. We evaluate these combinations using two molecular dynamics packages on a 512-core cluster and report up to a 25% increase in performance over the RC transport by using a software-level reliability layered over the UC transport.

The rest of the chapter is organized as follows: In Section 6.1 we describe our methodology to determine the costs of reliability. Section 6.2 provides our MPI design that we use for evaluation. The design implementation is described in Section 6.3 and the performance evaluation and analysis is in Section 6.4.

## 6.1 Methodology

We design our study to evaluate the tradeoffs associated with providing reliability within the network hardware and in the MPI library. In particular, we seek to evaluate the performance and resource requirements of the host for a variety of messaging patterns.

### 6.1.1 Native Performance

Before measuring software reliability, the native hardware performance for both reliable and unreliable communication modes must be evaluated. This baseline gives the basic

communication performance differences. In the case of InfiniBand, the Reliable Connection (RC) mode has likely been significantly more optimized than the Unreliable Connection (UC) mode due to its more prominent usage in libraries and applications. Since many modern networks provide link level flow control, packet loss is minimal and real-world workloads can be run over unreliable protocols for the purposes of evaluation. The time difference between the reliable and unreliable mode application executions of a workload represent the maximum improvement that can be obtained by using a software-based reliability mechanism.

### 6.1.2 Reliability Performance

The main evaluation criteria is the performance that can be obtained through providing reliability in the MPI library instead of hardware. To perform this evaluation, the same messaging workloads must be used on both a software-based implementation of reliability and a hardware implementation of reliability. To isolate this difference, the rest of the hardware and software stack should be identical.

Since InfiniBand provides both reliable and unreliable semantics on the same hardware it offers an ideal platform for our study. Current generation Mellanox InfiniBand adapters allow us to study three transports and isolate the reasons for performance differences:

- Reliable Connection (RC): Reliability and message segmentation are provided in hardware
- Unreliable Connection (UC): Message segmentation is provided in hardware
- Unreliable Datagram (UD): Neither reliability or segmentation is provided in hardware.

These three transports allow us to isolate the tradeoffs of providing hardware reliability and segmentation. In particular, comparing RC and UC gives the difference of hardware-level reliability and software-level reliability. The difference between RC and UC gives insight into the differences between connection-oriented and connection-less transports as well as hardware and software level segmentation.

### 6.1.3 Resource Usage

Another important metric to be measured is the additional memory usage incurred by the software to provide reliability support. To provide high performance message passing over an unreliable transport, significant buffering of unacknowledged messages may be required. The maximum memory usage for message buffering will be tracked. Additionally, the number of ACKs issued by the MPI library to provide reliability will also be tracked.

## 6.2 Proposed Design

The reliability protocol used has a significant influence on two of our evaluation metrics from Section 6.1 – performance and resource usage. In this section we discuss the suitable reliability protocols.

### 6.2.1 Reliability Engine

Earlier in Chapter 5 we have evaluated three possible reliability protocols for MPI over the UD transport of InfiniBand. In this work we select the "progress engine" style of reliability, which provided the best performance in the previous evaluation. More details on this can be found in our prior work.

In this method, shown in Figure 6.1, message segments are acknowledged lazily when the application makes calls into the MPI library. If there is a large skew between processes,

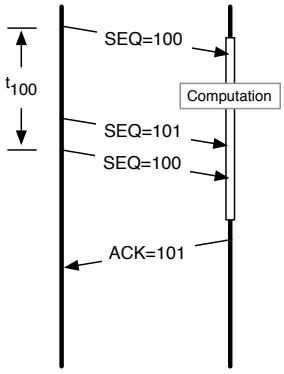



Figure 6.1: Progress-Based Acknowledgments

there is the possibility of unneeded message retries. Our previous work has showed this level to be acceptable. The next subsection gives the protocol details.

Note that additional reliability modes, particularly those based on kernel involvement, are not evaluated here. This work specifically targets providing reliability within the MPI library. It is anticipated that a kernel-level interrupt scheme may in some cases be able to provide even higher performance than the strictly user-level method that we are evaluating.

### 6.2.2 Reliable Message Passing

For small messages, those less than 8KB, our design uses the traditional sliding window protocol. The sender issues packets in order as there is available space in the window. In this manner the window represents the maximum number of unacknowledged message segments outstanding. Additional send operations occurring when the send window is already full are queued until outstanding operations have been acknowledged.

To enable reliability, after each message segment is sent it is tagged with a timestamp and added to an unacknowledged queue. This timestamp is provided through the Read Time

Stamp Counter (RDTSC) assembly instruction. If the MPI library detects that the timestamp has aged past a pre-configured timeout value ($t_{timeout}$) without an acknowledgment, the message segment is retransmitted. The receiver will discard all duplicate messages and send an explicit ACK to prevent additional retransmissions.

ACKs are also "piggybacked" onto all other messages being sent to the sender since many applications have bi-directional communication patterns. This significantly reduces the number of explicit ACKs that are required. If an ACK has not been piggybacked after $t_{timeout}/2$ usec, the receiver issues an explicit ACK to the sender.

Our design also uses ACKs for flow control in both the reliable and unreliable modes. This signaling is used to relay the number of free receive buffers available to the sender. Our reliability method also makes use of these control messages that are already being issued in the background.

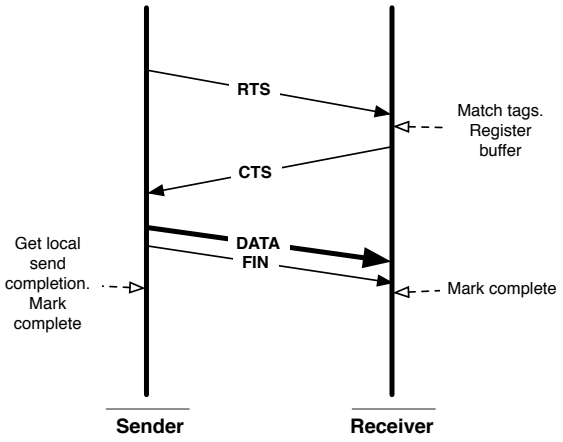
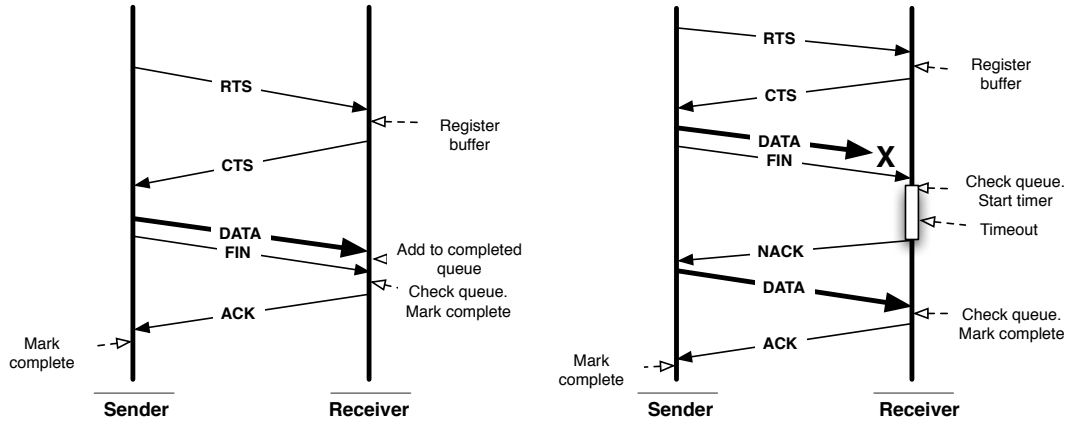



Figure 6.2: Traditional Zero-Copy Rendezvous over RC




(a) Non-Error Case (b) Retry Case

Figure 6.3: Zero-Copy over Unreliable Connection

### Preserving Zero-Copy over Unreliable Connection

On interconnects with RDMA capabilities, a zero-copy protocol is often used to transmit large messages. In zero copy protocols the sender and receiver use small control messages to match the message and then the message data is placed directly in user memory. A zero-copy protocol can significantly increase bandwidth and reduce cache pollution. Instead of performing data copies in the send and receive paths, within user-space or the kernel, a zero-copy approach directly sends the data from the source application buffer to the final destination buffer.

On InfiniBand, a handshake protocol (rendezvous) is used. As shown in Figure 6.2, the sending process sends a Request to Send (RTS) message with message tag information. Upon discovery of the message by the MPI layer at the receiver end, the receiver will send a Clear to Send (CTS) to the sender. If the sender receives a CTS, then it can use RDMA Write to send the message data directly to user application buffer at remote side. Thus,

RDMA Write provides a convenient method to perform zero-copy protocols when MPI message tags are matched by the MPI library.

To maintain these zero-copy semantics for an unordered and unreliable transport, this protocol must be adjusted. The traditional RDMA Write rendezvous protocol over RC makes use of both of the assumptions of ordering and reliability. A Finish (FIN) message is issued directly after the RDMA write data message and upon receipt the receiver knows the data has already arrived and can mark the receive as complete. Similarly, upon getting a local send completion from the CQ for the FIN message, the sender can mark the send complete.

Unfortunately, with an unreliable transport, neither of these mechanisms can be used. Since UC is unordered, the FIN message may arrive before the data is placed by the RDMA Write operation. Additionally, the local send completion on the sender no longer indicates successful data placement on the receiver.

To adapt the protocol for UC, the RDMA write operation must signal the receiver as well. InfiniBand has the option to send a 32-bit *immediate* data field with an RDMA write operation that generates a CQ entry on the target. We leverage this capability to allow the receiver to match the RDMA Write and FIN data messages. The FIN message is sent through the usual reliability channels with a sender-side retry – thus guaranteeing it will be received by the receiver. As seen in Figure 6.3(b), if the RDMA Write immediate data is not received within a timeout period, a NACK message is sent to the sender, which will trigger a resend of the RDMA Write data message. However, if the immediate data is received within the timeout a ACK message is immediately returned, allowing the sender to mark the send operation complete (Figure 6.3(a)).

It is important to note that the additional ACK requirement may decrease application performance in some cases. In particular for blocking `MPI_Send` operations, the sender cannot mark the send complete and proceed with application execution until the ACK is received.

## 6.3 Implementation

We have implemented our design over the verbs interface of the OpenFabrics/Gen2 stack [55].

We extend the UD-based ADI device described in Chapter 5 to include support for both RC and UC transports, as well as the zero-copy reliability design described earlier. All transports are incorporated into a single codebase to allow isolation of hardware performance characteristics and avoid software differences. UC and RC message handling code paths are identical aside from setup code. Since UC does not currently support Shared Receive Queue (SRQ), both UC and RC messaging layers are implemented without it.

Additionally, the library is instrumented to give insight into performance differences by tracking message statistics. Information such as the memory usage of the reliability protocol as well as message rates and sizes are tracked.

## 6.4 Evaluation and Analysis

In this section we evaluate each of the following combinations to determine the cost of reliability:

- **RC - Native**: RC transport with no software reliability

- **RC - Reliable**: RC transport with the software reliability layer enabled. Although in production such a combination would not be used, it allows us to observe the cost of reliability above the RC transport.

- **UC - Native**: UC transport with no software reliability. This is not a 'safe' configuration for production since message drops can occur. For research purposes, however, it gives an upper bound on the performance improvement that can be provided by a software reliability method.

- **UC - Reliable**: UC transport with software reliability.

- **UD - Native**: UD transport with no reliability. Same caveats and rationale as 'UC - Native' apply here.

- **UD - Reliable**: UD transport with software reliability.

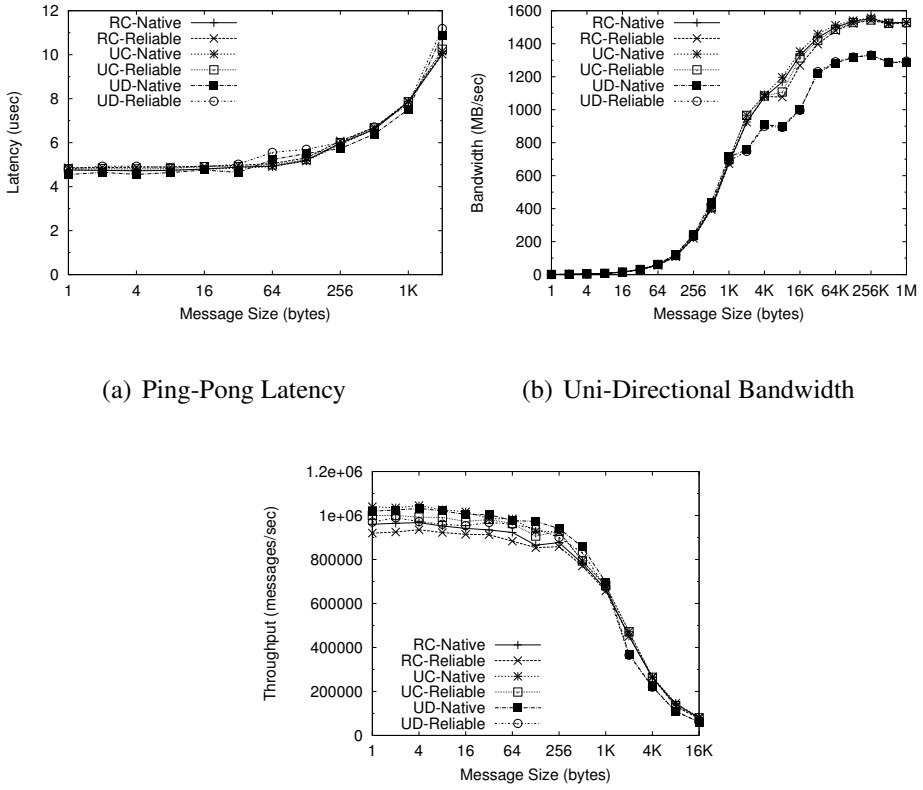
We evaluate first with microbenchmarks to observe the basic performance of each of the transports. Next we evaluate each of the combinations on two molecular dynamics applications, NAMD and LAMMPS.

### 6.4.1 Experimental Setup

Our experimental test bed is a 560-core InfiniBand Linux cluster. Each of the 70 compute nodes have dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of 8 cores per node. Each node has a Mellanox MT25208 dual-port Memfree HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55], OFED 1.2 release.

### 6.4.2 Microbenchmarks

In this section we evaluate each of the combinations on three microbenchmarks: ping-pong latency, uni-directional bandwidth, and uni-directional message rate.




(a) Ping-Pong Latency

(b) Uni-Directional Bandwidth

(c) Message Rate

Figure 6.4: Microbenchmark Transport Performance Comparison

*Ping-Pong Latency*: Figure 6.4(a) shows the latency of each of the transports with the software reliability toggled on and off. The latency for all transports is nearly identical. For messages under 64 bytes, UD-Native provides the lowest latency.

*Uni-Directional Bandwidth*: To measure the uni-directional bandwidth, we use the `osu_bw` benchmark [53], which sends a window of sends from one task to another and measures the time until an acknowledgment from the receiver is received. Figure 6.4(b) shows the results for each of the evaluation configurations. There is little difference between the UC and RC transports when natively used. The software reliability layer adds a noticeable but minimal decrease of 50 MB/sec in throughput between 8KB and 32KB. The UD transport shows lower performance for all message sizes above the MTU size since all messages above that level must be segmented by the MPI library instead of the HCA.

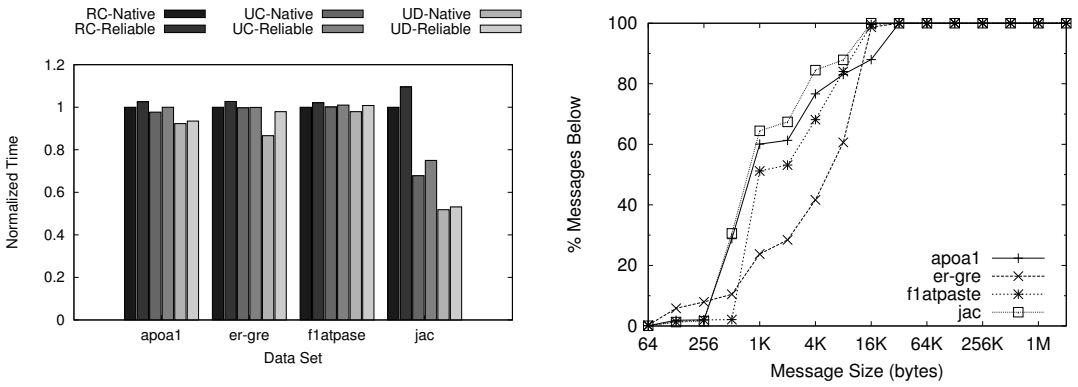
*Message Rate*: The message rate is measured in a similar manner as the uni-directional bandwidth. In this case, however, we report the number of messages sent per second instead of the bandwidth achieved. This shows the ability of the HCA to inject messages into the network. The results are shown in Figure 6.4(c). The results show that the unreliable transports, UC and UD, natively achieve nearly 10% higher throughput than RC. Even when layered with reliability the unreliable transports are able to outperform RC.

### 6.4.3 Application Benchmarks

In this section we evaluate each of the reliability combinations with two molecular dynamics applications. We evaluate with both NAMD and LAMMPS applications.

#### NAMD

NAMD is a fully-featured, production molecular dynamics program for high performance simulation of large biomolecular systems [59]. NAMD is based on Charm++ parallel objects, which is a machine independent parallel programming system. Of the standard data sets available for use with NAMD, we use the `apoa1`, `f1atpase`, `er-gre`, and `jac2000` datasets. We evaluate all data sets with 512 tasks.




(a) NAMD Normalized Time

(b) MPI Message Distribution

Figure 6.5: NAMD Evaluation Results

Figure 6.5(a) shows the results for each of the evaluation combinations and datasets. Characteristics of the application execution are provided in Table 6.1. In most of the cases UC-Reliable performs within 1% of RC-Native. Our software-based reliability adds 2%, 1%, and 1% of overhead above UC-Native for `apoa1`, `er-gre`, `f1atpase`, respectively – matching that of the hardware.

One particular dataset, the Joint Amber-Charm benchmark (`jac2000`), shows a large performance difference between each of the transports. Table 6.1 shows that there is a very high message rate used by this application. Per process there are nearly 7000 messages per second transmitted. Given this higher message rate the overhead caused by the reliability protocol is increased to 7% in the case of UC-Reliable. Even with the software reliability layer, however, the execution time is 25% reduced from RC-Native. The UD transport shows even higher performance, likely due to the lack of connection cache thrashing [79] in addition to the software reliability.

Additionally, as shown in Table 6.1, there is very little memory required to provide the software-layer reliability. Less than 1MB of memory is used per process at the high-watermark of memory usage for all datasets. The average maximum number of entries per process waiting for acknowledgments at any time was between 50 and 200. This value will depend on the application behavior and synchronization of the application. This number could be limited if needed, although it was not in this evaluation.

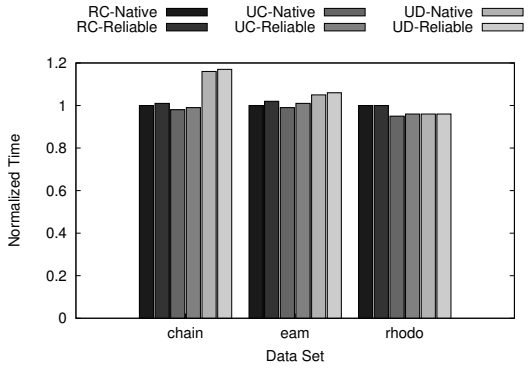
**LAMMPS**

Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [60] is a classical molecular dynamics simulator from Sandia National Laboratories. Several benchmark datasets are available from the LAMMPS website. They are meant to represent a range of simulation styles and computational expense for molecular-level interaction forces. In our experimental analysis, we used the Rhodospin protein (`in.rhodo`), Polymer chain melt (`in.chain`) and EAM Metal (`in.eam`) datasets available from the LAMMPS website. LAMMPS reports the "Loop Time" for a particular benchmark as a measure of time required to simulate a set of interactions.

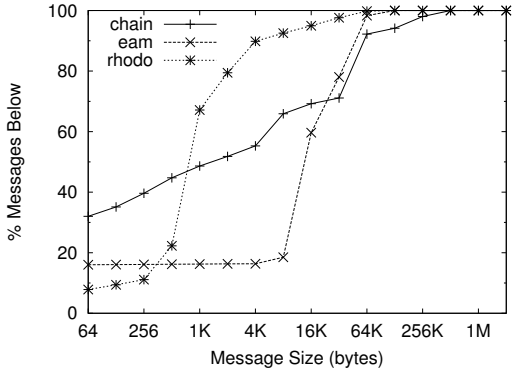
The results of our evaluation can be found in Figure 6.6(a). Characteristics of the communication are supplied in Table 6.2. As with NAMD, the performance between RC-Native and UC-Reliable is nearly identical in most cases. The software-based reliability layer for UC-Reliable adds a 1% and 2% overhead from UC-Native for `in.chain` and `in.eam`, respectively. The combinations with the UD transport perform significantly worse with these datasets, 17% and 6% worse than RC-Native. This is due to the segmentation costs incurred for the large messages in these datasets. As shown in Figure 6.6(b) 40% of the messages for `in.chain` and 80% of those for `in.eam` are over 8KB.

Table 6.1: NAMD Characteristics Summary (Per Process)

| Characteristics | | | | Data Sets | | | |
|---|---|---|---|---|---|---|---|
| | | | | apoa1 | er-gre | f1atpase | jac2000 |
| General | MPI Message Rate (msg/sec/process) | | | 1579.14 | 1958.28 | 2338.16 | 6979.33 |
| | MPI Volume Rate (MB/sec/process) | | | 6.27 | 13.43 | 9.16 | 15.89 |
| | Communicating Peers (peers/process) | | | 504 | 504 | 504 | 504 |
| Transport Specific | RC | Native | Flow ACKs | 234.60 | 216.02 | 565.85 | 193.63 |
| | | Reliable | Flow Control ACKs | 257.79 | 50.38 | 1717.47 | 26.06 |
| | | | Reliability ACKs | 43106.37 | 16786.32 | 118445.53 | 56299.19 |
| | | | Avg. Max Reliability Memory | 0.68 MB | 0.44 MB | 0.82 MB | 0.33 MB |
| | | | Avg. Max Queue Length | 124.01 | 71.34 | 170.45 | 65.08 |
| | UC | Native | Flow Control ACKs | 234.65 | 216.07 | 560.28 | 194.28 |
| | | Reliable | Flow Control ACKs | 210.32 | 41.79 | 1301.50 | 37.09 |
| | | | Reliability ACKs | 43149.78 | 16830.40 | 118914.76 | 57981.19 |
| | | | Avg. Max Reliability Memory | 0.64 MB | 0.47 MB | 0.79 MB | 0.35 MB |
| | | | Avg. Max Queue Length | 144.06 | 69.00 | 158.22 | 61.85 |
| | UD | Native | Flow Control ACKs | 287.88 | 103.82 | 801.51 | 301.94 |
| | | Reliable | Flow Control ACKs | 0.65 | 0.06 | 3.56 | 0.04 |
| | | | Reliability ACKs | 41897.84 | 16649.10 | 114419.89 | 57428.87 |
| | | | Avg. Max Reliability Memory | 0.22 MB | 0.11 MB | 0.41 MB | 0.12 MB |
| | | | Avg. Max Queue Length | 33.19 | 14.82 | 38.42 | 20.20 |




(a) LAMMPS Normalized Time




(b) MPI Message Distribution

Figure 6.6: LAMMPS Evaluation Results

Table 6.2: LAMMPS Characteristics Summary (Per Process)

| Characteristics | | | | Data Sets | | |
|---|---|---|---|---|---|---|
| | | | | chain | eam | rhodo |
| General | MPI Message Rate (msg/sec/process) | | | 372.34 | 332.73 | 4849.74 |
| | MPI Volume Rate (MB/sec/process) | | | 9.33 | 7.14 | 13.2 |
| | Communicating Peers (peers/process) | | | 15 | 8 | 90 |
| Transport Specific | RC | Native | Flow Lacks | 37.39 | 0.51 | 428.86 |
| | | Reliable | Flow Control ACKs | 11.34 | 0.12 | 54.53 |
| | | | Reliability ACKs | 1817.20 | 9310.43 | 28696.25 |
| | | | Avg. Max Reliability Memory | 0.08 MB | 0.04 MB | 0.13 MB |
| | | | Avg. Max Queue Length | 16.83 | 7.67 | 65.09 |
| | UC | Native | Flow Control ACKs | 37.26 | 0.53 | 428.86 |
| | | Reliable | Flow Control ACKs | 11.39 | 0.11 | 55.31 |
| | | | Reliability ACKs | 1814.69 | 8992.67 | 28670.16 |
| | | | Avg. Max Reliability Memory | 0.08 MB | 0.04 MB | 0.13 MB |
| | | | Avg. Max Queue Length | 16.73 | 7.84 | 65.30 |
| | UD | Native | Flow Control ACKs | 21.24 | 0 | 164.70 |
| | | Reliable | Flow Control ACKs | 0 | 0 | 0.19 |
| | | | Reliability ACKs | 1983.67 | 10443.20 | 29322.30 |
| | | | Avg. Max Reliability Memory | 0.07 MB | 0.07 MB | 0.12 MB |
| | | | Avg. Max Queue Length | 8.82 | 7.61 | 8.47 |

The `in.rhodo` dataset shows higher performance for both the unreliable transports, even after adding the software reliability layer. The MPI message rate of this dataset is nearly 5000 messages/sec, an order of magnitude higher than the other datasets. As seen in the NAMD `jac2000` dataset, the higher message rate seems to favor the unreliable transports.

As with NAMD, the memory required to support reliability at the MPI layer is minimal. Less than 128KB of memory was required at maximum for the evaluated datasets. A large number of explicit ACK messages were sent by the reliability layer, but this had minimal impact on performance.

**Analysis**

In both of the application runs it was observed that datasets where message rates were relatively low – less than 3000 messages/sec – the performance between UC and RC was similar and the software-based reliability protocols added little overhead.

When the message rate is increased the performance of the unreliable transports improves dramatically over RC. It is likely that HCA resources are becoming exhausted and the software-based approach that makes use of the host CPU and delayed ACKs leads to the improvement. This seems to be particularly prevalent when there are a large number of communicating peers.

In the case of the `jac2000` benchmark for NAMD there was a significant overhead to provide reliability in software. Upon further examination, it appears that the communication pattern is not as bi-directional as others. As a result, after controlling for execution time, the number of explicit (non-piggybacked) ACKs is nearly double that of other datasets (including that of `in.rhodo` for LAMMPS).

Our work shows that providing reliability at the MPI layer is not only feasible, but in some cases may provide higher performance. Furthermore, it can be done with little memory usage on the host.

## 6.5 Related Work

Other researchers have explored providing reliability at the MPI layer as part of the LA-MPI project [23, 6]. LA-MPI is focused on providing network fault-tolerance at the host to catch potential communication errors, including network and I/O bus errors. Their reliability method works on a watchdog timer with a several second timeout, focusing on providing checksum acknowledgment support in the MPI library. Saltzer et al. [65] discusses about the need for end-to-end reliability instead of a layered design.

Fast Messages (FM) [56] implements an ordering and reliability layer on top of Myrinet. This lightweight layer implements internal message buffering for performance and reliability.

Our work is different in that our focus is on evaluating the cost of reliability in hardware versus software. We believe this is the first such study where the hardware and software are fixed, with the only difference being the location of the reliability implementation. We leverage the work of other traditional reliability schemes to ensure a fair comparison.

# CHAPTER 7

# MULTI-TRANSPORT HYBRID MPI DESIGN

As InfiniBand clusters continue to expand to ever increasing scales, the need for scalability and performance at these scales remains paramount. As an example, the "Ranger" system at the Texas Advanced Computing Center (TACC) includes over 60,000 cores with nearly 4000 InfiniBand ports [81]. By comparison, the first year an InfiniBand system appeared in the Top500 list of fastest supercomputers was in 2003 with a 128 node system at NCSA [2]. The latest list shows over 28% of systems are now using InfiniBand as the compute node interconnect.

Its popularity growing, InfiniBand-based MPI benefits from ongoing research and improved performance and stability. The Message Passing Interface (MPI) [48] is the dominant parallel programming model on these clusters. Given the role of the MPI library as the communication substrate for application communication, the library must take care to provide scalability both in performance and in resource usage. As InfiniBand has gained in popularity, research has continued on improving the performance and scalability of MPI over it. Various optimizations and techniques have been proposed to optimize for performance and scalability, however, as InfiniBand reaches unprecedented sizes for commodity clusters it is necessary to revisit these earlier works and take the best aspects of each.

As an example, our previous studies have shown that the memory footprint for InfiniBand communication contexts can be significant at such large scales, impeding the ability to increase problem resolution due to memory constraints. A significant reason for this is the Reliable Connection (RC) transport used in most MPIs over InfiniBand, which requires a few KB of dedicated memory for each communicating peer. Our earlier proposed solution in Chapter 5 to this issue uses the Unreliable Datagram (UD) transport exclusively. Even this method has limitations, however, since the performance of UD is below that of RC in many situations, particularly for medium and large messages.

In this chapter we seek to address two main questions:

- What are the current message channels developed over InfiniBand and how do they perform with scale?
- Given this knowledge, can an MPI be designed to dynamically select suitable transports and message channels for the various types of communication to improve performance and scalability for the current and next-generation InfiniBand clusters?

As part of this work we develop a multi-transport MPI for InfiniBand, MVAPICH-Aptus[2], which dynamically selects the underlying transport protocol, Unreliable Datagram (UD) or Reliable Connection (RC), as well as the message protocol over the selected transport on a per message basis to increase performance over MPIs that only use a single InfiniBand transport. We design flexible methods to enable the MPI library to adapt to different network and applications.

Our results on a variety of application benchmarks are very promising. On 512 cores, MVAPICH-Aptus shows a 12% improvement over an RC-based design and 4% better than

[2] 'Aptus' is Latin for "appropriate or fitting"

a UD-based design for the SMG2000 [15] application benchmark. In addition, for the molecular dynamics application NAMD [59] we show a 10% improvement over an RC-only design.

The rest of the chapter is organized as follows: We give a brief overview of all available channels in Section 7.1. A variety of microbenchmarks and evaluations are used to examine the performance and overall scalability of each message channel in Section 7.2. In Section 7.3 we propose our framework, "Aptus," that allows multiple message channels to be used simultaneously for increased performance. We evaluate our design in Section 7.4.

## 7.1 Message Channels

In this section we describe each of the communication channels that are available for message transfer in an InfiniBand-based cluster. As described in Section 2.2, MPI designs typically use two protocols: eager and rendezvous. We briefly summarize all of the communication channels that have been used previously in MVAPICH as well as those designed as part of this thesis.

### 7.1.1 Eager Protocol Channels

*Reliable Connection Send/Receive (RC-SR)*: We refer to RC-SR as the channel built directly on the channel semantics of InfiniBand. It is the primary form of communication for small messages on nearly all MPI implementations over InfiniBand. Two designs have been proposed, one with per-peer credit-based flow control and the other using the Shared Receive Queue (SRQ) support of InfiniBand. In this work we use only the SRQ-based design since it has superior scalability (detailed and shown in earlier work [75, 69]), and

since it allows receive buffers to be pooled across QPs (connections) instead of posted on a per-peer basis.

*Reliable Connection Fast-Path (RC-FP)*: Current InfiniBand adapters only reach their lowest latency when using RDMA write operations, with channel semantics having a 2$\mu$sec additional overhead (e.g. 5$\mu$sec vs. 3$\mu$sec) on our evaluation hardware. The newest Mellanox adapter, ConnectX [47], reduces this gap to less than a microsecond, however RDMA write operations still achieve the lowest latency [77].

To leverage this capability, small message transfer has been designed over the RDMA write mechanism to facilitate the lowest latency path of communication [42]. Dedicated buffers are required for each communicating peer – the default MVAPICH configuration requires over 300KB of memory per RC-FP channel created. To limit memory usage, channels are currently setup adaptively and limited to a configurable number of channels in current MPIs over InfiniBand. In addition, each RC-FP channel requires polling an additional memory location for detection of message arrival. For example, communication with $n$ peers using the RC-FP channel requires polling $n$ memory locations for message arrival.

*Unreliable Datagram Send/Receive (UD-SR)*: As designed and described in Section 5.2, the UD-SR message passing channel is message transfer implemented over the channel semantics of the UD transport of InfiniBand. Message segmentation and reliability must be handled within the MPI library to provide the guarantees made by the MPI specification to applications. Advantages of using this channel include superior memory utilization since a single UD QP can communicate with any other UD QPs; each QP is not dedicated to a specific peer as with the RC transport.

### 7.1.2 Rendezvous Protocol Channels

*Reliable Connection RDMA (RC-RDMA)*: The RC-RDMA channel is the mechanism for sending large messages. Using this method, the sender can use an RDMA write operation to directly write into the application buffer of the receiver without intermediate copies. Additional modes have also been suggested based on RDMA read [76] and a pipelined RDMA write [72]; however, in this work we consider only RDMA write.

*Unreliable Datagram Zero-Copy (UD-ZCopy)*: In Section 5.5, a zero-copy protocol for transferring large messages over the UD transport of InfiniBand was proposed. Bandwidth for large messages is significantly increased due to the lack of copies on both sides of communication. The primary motivation for this channel is to provide high bandwidth and to avoid scalability problems with RC QPs.

*Copy-Based Send*: If neither of the previously noted rendezvous channels are available, large messages can be segmented within the MPI library into many small sends and sent using an eager protocol channel (after negotiating buffer availability). This method, however, introduces intermediate copies and degrades performance.

### 7.1.3 Shared Memory

Clusters with multiple tasks running per node often use shared memory communication to communicate within a single node. This reduces contention on the network device and can provide lower latency and higher performance. In this work we will consider the design described in [17], which is included in current versions of MVAPICH [53]. This provides both an eager and rendezvous protocol design for intra-node communication.

## 7.2 Channel Evaluation

Our experimental test bed is 560-core InfiniBand Linux cluster. Each of the 70 compute nodes has dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of 8 cores per node. Each node has a Mellanox MT25208 dual-port Memfree HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55], OFED 1.2 release.

The RC-based message channels we evaluate follow the design of MVAPICH [53]. The UD-based message channels are based on the design included in MVAPICH-UD, described in Chapter 5.

We evaluate the basic performance of latency and bandwidth of the channels, followed by an investigation into the scalability of each channel.

### 7.2.1 Basic Performance Microbenchmarks

In this section we investigate the basic characteristics of each message passing channel that is available.

*Ping-Pong Latency*: Figure 7.1 shows the latency of each of the eager message channels. RC-FP shows the lowest latency, with a minimum of slightly less than 3$\mu$sec. RC-SR and UD-SR have very similar latency results up to 2KB; at this point UD-SR is sending 2KB of data as well as the required MPI header information – resulting in a message size of over 2KB, which is the MTU on our evaluation HCA. This requires segmentation within the MPI library due to limits of the UD transport; this cost is clearly visible.

*Uni-directional Bandwidth*: To measure the uni-directional bandwidth, we use the `osu_bw_mr` benchmark [53], which sends a window of sends from one task to another and measures the time until an acknowledgment from the receiver is received. In addition, the benchmark

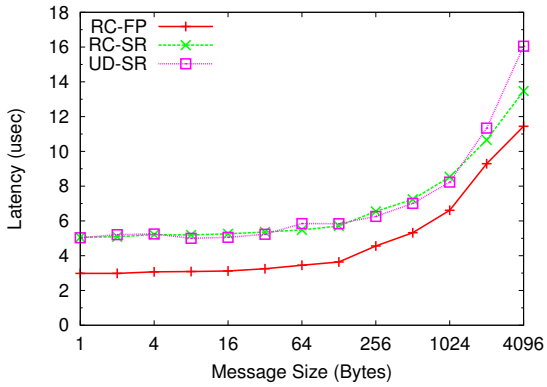
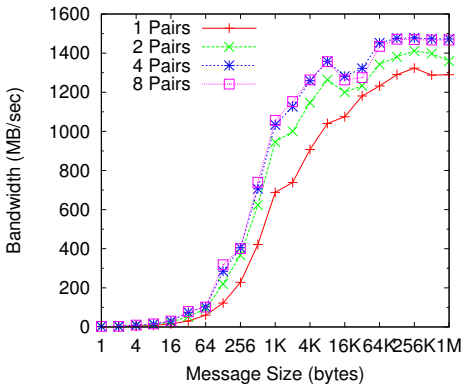
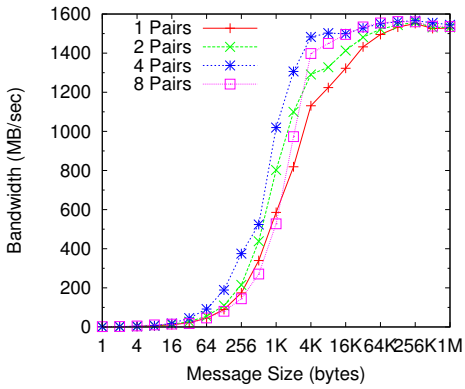


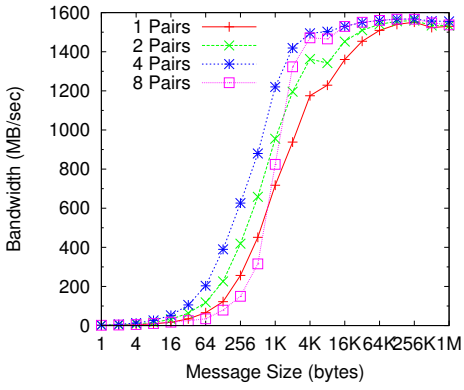

Figure 7.1: Channel Latency Comparison



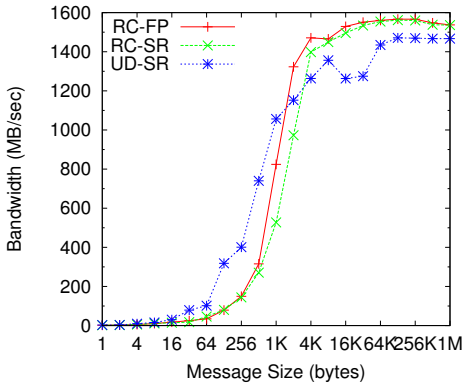

(a) UD-SR/UD-ZCopy




(b) RC-SR/RC-RDMA




(c) RC-FP/RC-RDMA




(d) 8 Pairs

Figure 7.2: Multi-Pair Uni-Directional Bandwidth Comparison

measures the aggregate bandwidth achieved by multiple pairs of communicating tasks between nodes.

Figure 7.2 shows the results for each of the eager message channels, paired with the rendezvous channel of the same transport. For small message sizes we see that UD-SR demonstrates high bandwidth and there is no decrease in performance with additional pairs of tasks. By contrast, RC-FP and RC-SR show performance degradation from 4 to 8 pairs of concurrent communication. The UD transport requires less HCA overhead since hardware-level ACKs are not sent for UD messages and state information does not need to be retained on the HCA. As in the latency evaluation, we note a decrease in performance for UD-SR after 1KB due to segmentation overhead.

For large message bandwidth we note that UD-ZCopy achieves significant throughput but is slightly lower than RC-RDMA for a single pair. Additional overheads, such as posting in 2KB chunks, are required in the UD-ZCopy protocol that lower the performance below the fabric limits that RC-RDMA achieves.

### 7.2.2 Evaluation of Channel Scalability

In this section we evaluate several other characteristics of each message channel, in particular those that have scalability aspects.

*Memory Footprint*: While a channel may provide high-performance it may come only at the cost of host memory usage. Figure 7.3(a) shows our measurement of channel memory usage per task. From the graph we immediately note that RC-FP consumes significant amounts of memory, making a large number of RC-FP channels infeasible. RC-SR/RC-RDMA also have a significant memory footprint as the number of connections increases since RC-SR/RC-RMDA are built on the RC transport. Recall from Section 2.1 that each

RC QP must be dedicated to another RC QP in the peer task. Memory usage for UD-based transports is negligible since a single UD QP can communicate with any other UD QP in any task, leading to superior memory scalability.

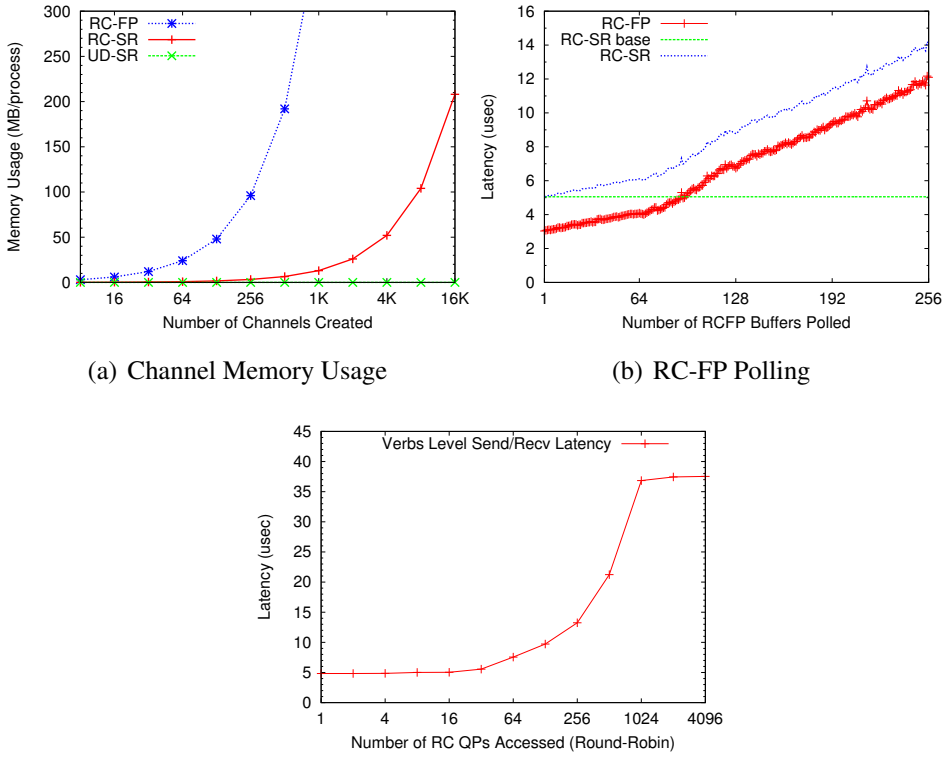
*Performance with Increasing Channel Count*: Another important aspect to evaluate is the effect of multiple channels on performance. In the earlier subsection we evaluated only two tasks, which does not show scalability related effects.

As described earlier, RC-FP requires dedicated receive buffers for each communicating peer. As a result, a byte for each RC-FP channel must be polled to detect message arrival. To explore the cost of this polling we modify the RC-FP channel to poll on a configurable numbers of buffers. Figure 7.3(b) shows the 4-byte latency with increasing poll counts. We also plot the line for RC-SR latency, since polling RC-FP buffers also delays the detection of messages arriving on any other channel. Based on this result it becomes clear that more than 100 RC-FP channels will lead to performance degradation over a RC-SR only design.

By contrast, RC-SR and UD-SR maintain the same latency as the number of allocated channels increases. All completions are placed in a single CQ, where the library can poll for message arrival.

*Impact of HCA Architecture*: Although RC-SR shows similar performance to UD-SR with increasing numbers of allocated channels, performance differs from UD-SR when each of the channels is *used* instead of simply allocated.

InfiniBand HCAs cache QP information using on-card memory, called the InfiniBand Context Memory (ICM) cache. The ICM cache has a limited size and cannot hold more than a limited number of QP entries at any one time; context information outside of the cache must be fetched from host memory.




(a) Channel Memory Usage

(b) RC-FP Polling

(c) Effect of HCA ICM Cache

Figure 7.3: Channel Scalability Evaluation

Table 7.1: Channel Characteristics Summary

| Type | Channel | Transport | Latency | Throughput | Scalability |
|---|---|---|---|---|---|
| Eager | RC Send/Receive (RC-SR) | RC | Good | Fair | Fair |
| | RC Fast-Path (RC-FP) | RC | Best | Good | Poor |
| | UD Send/Receive (UD-SR) | UD | $<$ 2KB, Good<br>$\geq$ 2KB, Poor | $<$ 2KB, Best<br>$\geq$ 2KB, Poor | Best |
| Rendezvous | RC-RDMA | RC | - | Best | Fair |
| | UD Zero-Copy (UD-ZCopy) | UD | - | Good | Best |
| | Copy-Based | UD or RC | - | Poor | - |

We replicate the evaluation from [79] to measure the cache size for our newer-generation HCAs by evaluating the 4-byte latency at the lowest software layer, the InfiniBand "verbs." Figure 7.3(c) shows that the ICM cache of the HCA is still limited in size with large increases in latency when multiple QPs are accessed in a round-robin order. All protocols based on the RC transport have this issue. Furthermore, this problem is exacerbated by the increase in core counts which lead to larger number of tasks sharing the same HCA (and ICM cache). UD-SR does not have this issue since a single UD QP can communicate with any other number of UD QPs – thus remaining in the HCA cache.

## 7.3 Proposed Design

In this section we describe our multi-transport design that incorporates all available communication channels. Since neither the RC or UD transport provides all of the desired features – scalability and best performance – a hybrid of the two transports is required. In this section we propose our design, MVAPICH-Aptus, that encompasses all of the available channels into a unified design that allows flexibility in channel selection. Based on

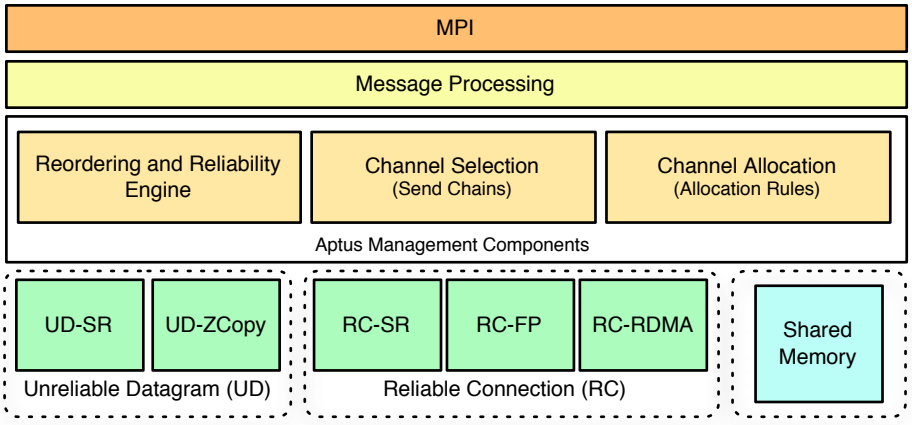



Figure 7.4: MVAPICH-Aptus Design Overview

the results from the previous section, a summary of channel characteristics is provided in Table 7.1. Figure 7.4 shows the general overview of the design.

We first describe the initial state of Aptus at startup, followed by a discussion of how Aptus multiplexes channels and provides reliability. Next we explain the channel selection algorithm of Aptus and the channel allocation strategies used.

### 7.3.1 Initial Channel Allocation

At job startup, `MPI_Init`, Aptus only creates UD QPs and exchanges their information to the other tasks in the job. At this point all tasks in the job are able to communicate with any other task using the UD-SR and UD-ZCopy channels. If tasks are sharing the same physical host, the SMP channel is also automatically allocated.

After communication begins, Aptus tracks communication characteristics to determine the peers that each task communicates most frequently with as well as the message sizes. Using these statistics Aptus dynamically allocates more resource intensive channels, such as RC-SR or RC-FP, to reduce the communication overhead between sets of peers. The

interpretation of these statistics must be done in consideration with the architecture and characteristics of each channel. Details are provided later in Section 7.3.4.

In addition, since performance characteristics for different message sizes vary with the channel, multiple channels between tasks may be required to obtain the best performance. For this reason Aptus allows for multiple channels to be active between tasks.

### 7.3.2 Channel Multiplexing and Reliability

As mentioned earlier, Aptus allows for multiple communication channels to be active simultaneously. As part of the support for UD-SR, Aptus must contain a reliability and re-ordering engine since the UD transport that underlies UD-SR is unreliable and subject to message drops and reordering. To support this, we design a generalized framework to re-order messages from all message channels since depending on the channel there may be out-of-order receives. In addition, each message channel can be independently configured to use the message reliability support. For example, is is not necessary for reliability to be enabled for RC-SR or RC-FP, so we can disable reliability. This elegant integration also allows other features such as an end-to-end CRC check to be done across all channels very simply if reliability is turned on for all channels and a CRC is computed on send and receive.

### 7.3.3 Channel Selection

Given the framework described, one area left unresolved is how to determine which message channel should be used of those allocated. The factors that motivate this selection are almost entirely from the architecture characteristics. Factors such as the different overheads between RC-SR and RC-FP may change with the HCA model and should be reflected in the way messages are sent. Our main observation is that factors that drive

message channel selection are fluid – they may change based on cluster architecture or the number of ranks in the MPI job.

To support a flexible model we design a *send rule chain* method of determining how messages should be sent. A single send rule is in the form of {`COND, MESG_CHANNEL`}, e.g. {`MSG_SIZE <= 1024, UD-SR `}. If `COND` evaluates to true and `MESG_CHANNEL` is already allocated, then `MESG_CHANNEL` will be used for this message. Multiple of these send rules can be chained together, with earlier rules taking priority over later rules in the chain. The last rule in the chain must have a conditional of `TRUE` with UD-based channel to be valid.

Based on our evaluation in Section 7.2, we found that RC-FP has superior latency and should be used if available. In addition, RC-SR and UD-SR perform similarly in latency for small messages, however UD-SR has better throughput for messages under 2KB. Larger messages always gain performance using RC-RDMA if available. In keeping with these findings, we develop the following default send rule chain for our system:

```
{ MSG_SIZE <= 2048, RC-FP }, { MSG_SIZE <= 2008, UD-SR },
{ MSG_SIZE <= 8192, RC-SR }, { MSG_SIZE <= 8192, UD-SR },
{ TRUE, RC-RDMA }, { TRUE, UD-ZCOPY }
```

[*Note that these rules do not take into account whether a channel should be created, just whether to use it if it has* ***already been allocated.***]

Using this flexible framework, send rules can be changed on a per-system or job level to meet application and hardware needs without changing code within the MPI library and re-compiling.

Although our current prototype does not support any other variables besides `MSG_SIZE` and `NUM_RANKS` in the conditional, other rules could potentially be designed to allow for additional flexibility.

### 7.3.4 Channel Allocation

The previous section explored how messages can be sent over different channels using a configurable send rule chain. In this section we discuss how each of those channels is allocated initially.

As discovered earlier in our channel evaluation, it is detrimental to performance as well as the memory footprint to create certain channels after a number of them have already been created. For example RC-FP adds latency to all other channels as well as itself as more peers are communicated with over RC-FP. Similarly, too many RC connections use a significant amount of memory and can overflow the HCA ICM cache, leading to significant latency.

It is important to use the minimal number of these channels while allocating high performance channels only to those peers that will benefit most from them. In our current prototype we use a variation of the send rules described in the previous section to increment counters to determine which peers would benefit most from a new connection. After the counter for a rule has reached its configured limit, the channel of that type is created provided per-task limits have not been passed.

Based on our evaluation, in our configuration we limit the number of RC-SR/RC-RDMA channels to 16 per task, meaning a task can only use the RC transport to a maximum of 16 of its peers. Similarly we limit RC-FP to 8 channels per task. These limits represent a tradeoff between the performance provided by each channel and the performance degradation that occurs with too many channels of these types. These limits are run-time tunable to allow flexibility based on HCA type and architecture. Limiting the RC QPs limits the potential for HCA cache thrashing.

## 7.4 Application Benchmark Evaluation

In this section we evaluate a prototype of our Aptus design on the NAS Parallel Benchmarks, NAMD, and SMG2000. Our evaluation platform is the same as described in Section 7.2. To explore performance changes based on the hybrid model of Aptus, we evaluate four different configurations:

- *RC*: In this configuration we evaluate using MVAPICH 0.9.9, in the default configuration, aside from additional receive buffers posted to the SRQ. This is the baseline performance expected for an RC-based MPI. Includes RC-SR, RC-FP, and RC-RDMA.
- *UD*: Our design using only UD-SR and UD-ZCopy (MVAPICH-UD)
- *UD-copy*: Our design using only UD-SR
- *Aptus*: The prototype of our design presented in this work with the parameters mentioned in Section 7.3.3 using UD-SR, UD-ZCopy, RC-SR, RC-FP, and RC-RDMA.

In addition, each of the configurations uses the same shared memory channel for all communication to peers on the same node.

In terms of channel designs, our Aptus prototype is based on MVAPICH-UD and MVAPICH, however, the codebase itself is almost entirely new. It is implemented as a device layer for MPICH.

For our evaluation we collect message statistics for each message channel. We track both the number of messages sent and the data volume sent over each channel. In addition, we track this on a per-peer basis to determine how many message channels are allocated and to what peers. We also determine the message distribution of the application from within the MPI library, including control messages. Figure 7.5 shows these distributions

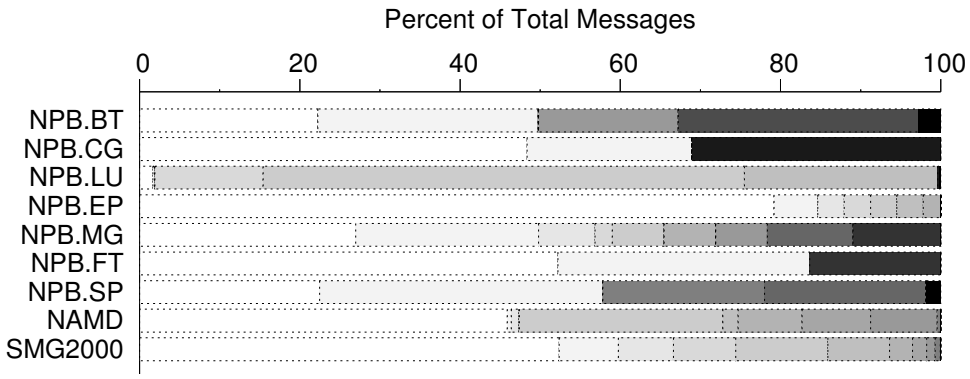



Figure 7.5: Message Size Distribution: Darker blocks denote larger message sizes (Brightness Index: 100%: $\leq$ 64 bytes, 50% $\leq$ 4KB, 25% $\leq$ 64KB , 0% $\geq$ 512KB)

with darker blocks denoting larger messages. For example, 95% of messages for LU are greater than 512 bytes, however, very few are greater than 4KB.

### 7.4.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks [8] (NPB) 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. We evaluate using the largest of the problem datasets, Class 'D'. We run CG, EP, FT, LU, and MG with 512 tasks and both BT and SP with 529 tasks.

Figure 7.6 shows execution time normalized to RC of the NAS Benchmarks. In each case the Aptus prototype maintains equal or better performance than the other configurations. We note that in general UD-Copy performs the worst since large messages incur intermediate copy overheads.

For both CG and LU the RC configuration outperforms that of UD. Figure 7.7(b) shows the percentage of messages sent over each message channel. We observe that over 40% of messages are able to be transferred over the low-latency RC-FP channel, which is not available in a UD-only implementation. For CG, we note from Figure 7.5 that over 30%

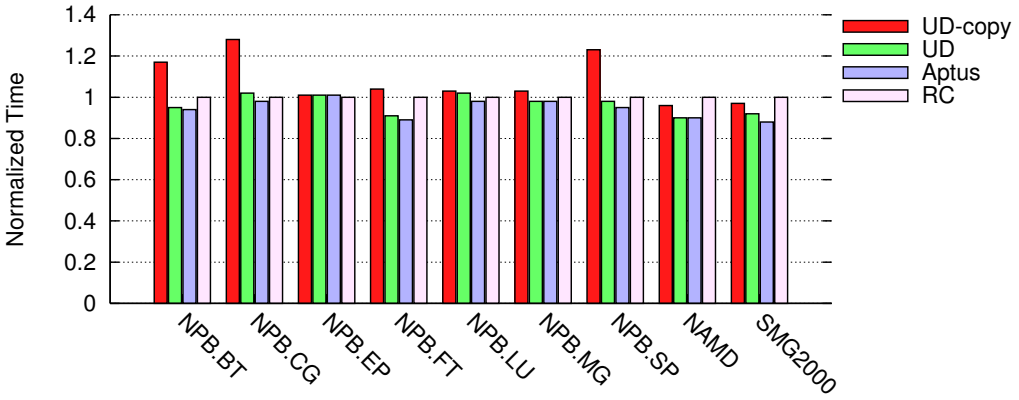
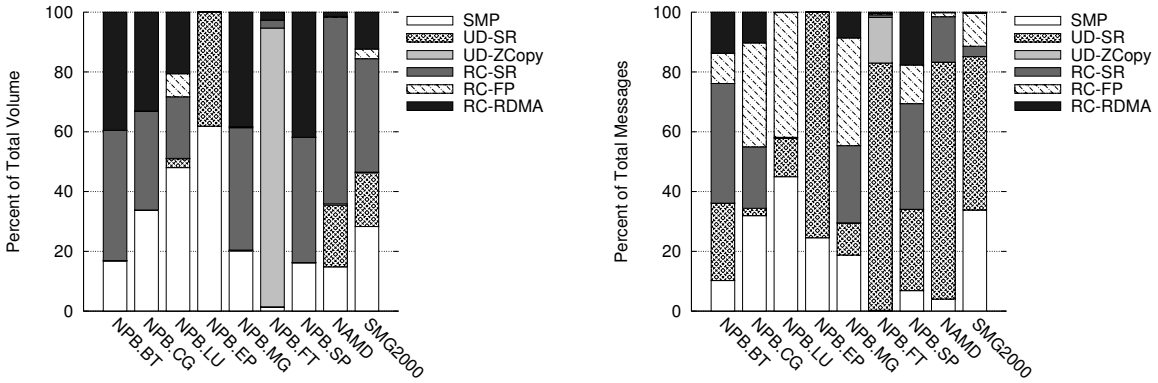



Figure 7.6: Application Benchmark Performance

Table 7.2: Average Number of Channels Used/Allocated Per Task (Aptus)

| App. | Message Channels | | | |
|---|---|---|---|---|
| | SMP | UD-{SR,ZCopy} | RC-{SR,RDMA} | RC-FP |
| NPB.BT | 4.11 | 20.17 | 10.60 | 7.88 |
| NPB.CG | 3.00 | 6.94 | 2.94 | 2.94 |
| NPB.EP | 3.00 | 6.00 | 0.00 | 0.00 |
| NPB.FT | 7.00 | 504.00 | 16.00 | 8.00 |
| NPB.MG | 4.31 | 9.00 | 5.63 | 5.63 |
| NPB.LU | 3.75 | 7.06 | 2.23 | 2.23 |
| NPB.SP | 4.11 | 20.17 | 10.62 | 7.88 |
| NAMD | 6.30 | 120.80 | 16.47 | 8.00 |
| SMG2000 | 4.25 | 120.19 | 16.34 | 8.00 |




(a) Data Transfer Volume

(b) Number of Messages

Figure 7.7: Aptus Channel Usage Distributions

of messages are over 256KB, where RC-RDMA can provide a significant benefit. Aptus takes the benefits of both RC-FP for small messages and RC-RDMA for large messages, while using UD-SR for less frequently communicating peers. As a result, for both LU and CG, Aptus performs 2% better than RC and 4% better than UD.

In other benchmarks, FT in particular, we note that UD outperforms the RC configuration. From Table 7.2 we note that FT performs communication with all peers and in the RC case will require 504 RC QPs, leading to QP thrashing as the HCA ICM cache is exceeded. By comparison, the UD QPs in the UD configuration will remain in the ICM cache and lead to increased performance, 9% in this case. Aptus shows a small improvement over UD since a few large messages can be sent using RC, reducing the load on the host.

### 7.4.2 NAMD

NAMD is a fully-featured, production molecular dynamics program for high performance simulation of large biomolecular systems [59]. 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 `f1atpase`. We evaluate with 512 tasks.

From Figure 7.6 we observe that NAMD performs much better (10%) using the UD transport than the RC transport. From Table 7.2 we observe that each NAMD task communicates with a large number of peers (120.8) on average. With so many communicating peers RC will overflow the ICM cache. In addition, many of the messages are below the 2KB segmentation threshold for UD-SR where performance exceeds that of RC-SR. Aptus performs similarly to UD with a 10% improvement over RC, which can be explained by Figure 7.7(b), which shows nearly 80% of messages are sent over UD-SR, meaning Aptus receives little benefit from RC-RDMA or RC-FP.

Note that although we have set the threshold for the number of RC-SR/RC-RDMA channels to 16, both NAMD and SMG2000 have an average above 16. This is due to the handshake involved in connection setup and the fact that both sides must create the connection. This is a benign race condition that can lead to a maximum of two additional connections.

### 7.4.3 SMG2000

SMG2000 [15] is a parallel semi-coarsening multigrid solver, which is part of the ASC Purple Benchmarks. We run SMG2000 for 512 tasks and report the solve time produced by the benchmark.

Figure 7.6 shows the execution time of SMG2000 normalized to RC, where we observe clear differences between each of our configurations. Aptus delivers the highest performance, a full 12% over RC and 4% over UD. As with NAMD, from Table 7.2, SMG2000 communicates on average with over 120 peers of the 512 in the job. Such a large number of peers favors the UD transport, which we observe in the increased performance of UD over RC. Aptus, however, further improves on the performance of UD by using RC-SR and RC-RDMA for larger messages. From Figure 7.7(a) we observe Aptus is able to send 50% of data volume over one of the RC-based message channels, which have better bandwidth than UD-SR for messages over 2KB.

## 7.5 Related Work

MPIs that dynamically use all available interconnects have been designed in the past, including Intel MPI, Open MPI, Scali MPI, and others. Research work has been done with Mad-MPI [7] from the Madeleine project that seeks to incorporate multiple interconnects and provides scheduling across them. Our work is different in that we are targeting different transports on the same network device, and optimizing for the memory footprint and performance.

Current MPI designs over InfiniBand such as MVAPICH, Open MPI, and Intel MPI offer dynamic creation of RC QPs as needed; however, none of them include support for both the UD and RC transports simultaneously and cannot limit the number of RC QPs that are created. If a peer communicates with all others in the job a QP will be created to each one. Our design by contrast allows the amount of allocated resources to be limited.

To the best of our knowledge, our work is the first to show the combination of both the RC and UD transports.

# CHAPTER 8

# SCALABLE MPI OVER EXTENDED RELIABLE CONNECTION

Current implementations of MPI over InfiniBand, such as MVAPICH, Open MPI, HP MPI, and others, use the Reliable Connection (RC) transport of InfiniBand as the primary transport. Earlier work has shown, however, that the RC transport requires several KB of memory per connected peer, leading to significant memory usage at large-scale. MVAPICH can also support the Unreliable Datagram (UD) transport for communication (Chapter 5), however, implementing MPI over UD requires software-based segmentation, ordering and re-transmission within the MPI library. Neither of these transports are ideal for MPI on large-scale InfiniBand clusters.

The latest InfiniBand cards from Mellanox include support for a new InfiniBand transport – eXtended Reliable Connection (XRC). The XRC transport attempts to give the same feature set of RC while providing additional scalability for multi-core clusters. Instead of requiring each process to have a connection to every other process in the cluster for full connectivity, XRC allows a single process to require only one connection per destination node. Given this capability, the connection memory required can potentially reduce by a factor equal to the number of cores per node, a potentially large degree as core counts continue to increase.

In this chapter we design MPI over the XRC transport of InfiniBand and discuss the connection requirements and opportunities it offers. An implementation of our design is evaluated using standard MPI benchmarks and memory usage is also measured. Application benchmark evaluation shows a 10% speedup for the `jac2000` NAMD dataset over an RC-based implementation. Other benchmarks show increased memory scalability but near-equal performance using the XRC transport. For a 16-core per node cluster, XRC shows a nearly 100 times improvement in connection memory scalability over a similar RC-based implementation.

This chapter is organized as follows: More details on the XRC transport of InfiniBand are described in Section 8.1. We present our XRC designs in Section 8.2. Evaluation and analysis of an implementation of our designs is covered in Section 8.3. This is followed by a discussion of work related to ours in Section 8.4.

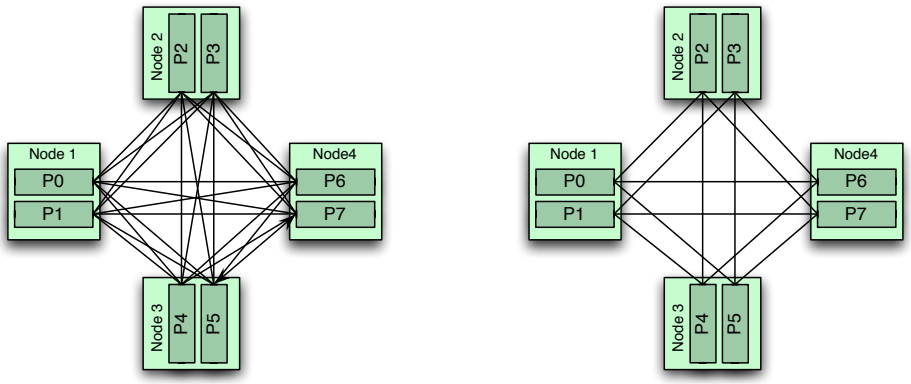
## 8.1 eXtended Reliable Connection

In this section we describe the new eXtended Reliable Connection (XRC) transport for InfiniBand. We first start with the motivation for this new transport followed by the XRC connection model and addressing.

### 8.1.1 Motivation

The motivation for creating the XRC transport comes from the explosive growth in multi-core clusters. While node counts continue to increase, core counts are increasing at an even more rapid rate. The Sandia Thunderbird and TACC Ranger show this trend:

The Sandia Thunderbird [67] was introduced in 2006 with 4K compute nodes each with dual CPUs for a total of 8K processing cores. The TACC Ranger [81] was put into




(a) RC Connection Model

(b) XRC Connection Model

Figure 8.1: InfiniBand Reliable Connection Models

production in 2008 with nearly 4K compute nodes. Each compute node has four quad-core CPUs, for a total cluster size of nearly 64K processing cores. Each at the time of introduction were in the upper echelon of the fastest machines in the world.

Existing InfiniBand transports made no distinction between connecting a process (generally one per core for MPI) and connecting a node. Thus, the associated resource consumption increased directly in relation to the number of cores in the system. Our earlier investigation in Chapter 4 showed that memory usage for the RC transport can reach hundreds of MB of memory/process at 16K processes.

To address this problem XRC was introduced. Instead of having a per-process cost, XRC was designed to allow a single connection from one process to an entire node. In doing so, the maximum number of connections (QPs) per process can grow with the number of nodes instead of the number of cores in the system.

### 8.1.2 Connection Model

XRC provides the services of the RC transport, but defines a very different connection model and method for determining data placement on the receiver in channel semantics.

When using the RC transport of InfiniBand, the connection model is purely based on processes. For one process to communicate with another over InfiniBand, each side must have a dedicated QP for the other. There is no distinction as to the node in terms of allowing communication.

Figure 8.1(a) shows a fully-connected job, with each node having two processes, each fully connected to the other processes on other nodes. To maintain full connectivity in a cluster with $N$ nodes and $C$ cores per node, each process must have $(N-1) \times C$ QPs created. In this figure and equation we do not account for intra-node IB connections since the focus of this work is on MPI and libraries generally use a shared-memory channel for communication within a node instead of network loopback.

By contrast, XRC allows connection optimization based on the location of a process. Instead of being purely process-based, the node of the peer to connect to is now taken into account. Consider a situation where a process $A$ on $host1$ wants to communicate with both process $B$ and process $C$ on $host2$. After $A$ creates a QP with $B$, $A$ is also now connected to process $C$. The addressing required is discussed in the next section. The additional complication here is that although $A$ can now send to $C$, it is not reciprocal since $C$ cannot send to $A$. To send a message, a process must have one XRC QP to the node of the destination process and in our example $B$ has the QP to $A$ (and can send to $A$). Thus, if $C$ wants to send to $A$ it would need to create a QP to $A$. Note, if $C$ had a QP to a process $D$ on the same node as $A$ it would be able to communicate with $A$.

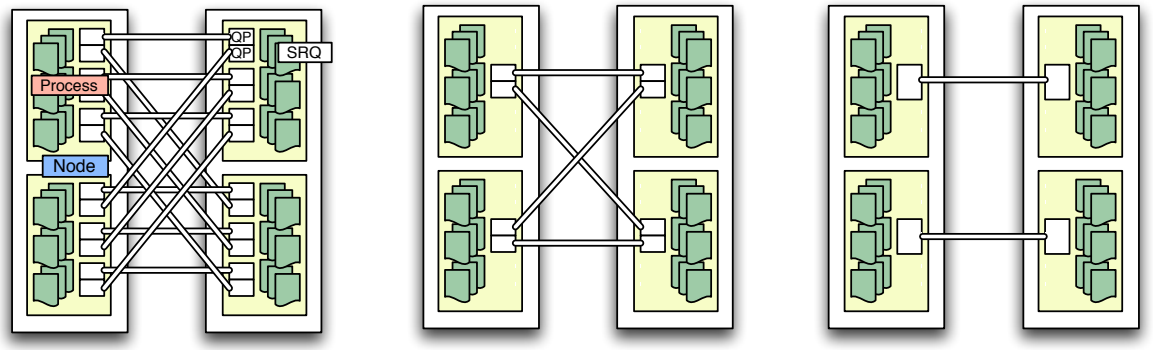
Figure 8.1(b) shows a fully-connected XRC job. Instead of requiring a new QP for each process, now each process needs to only have one QP per node to be fully connected. In the best case the number of QPs required for a fully-connected job in a cluster with $N$ nodes and $C$ cores per node, is only $N$ QPs. This reduces the number of QPs required by a factor of $C$, which is significant as the number of cores per node continue to increase.

### 8.1.3 Addressing Method

In the past, when using RC each process would communicate with a peer using a dedicated QP. Recall from Section 2.1, there are two forms of communication semantics: channel and memory. In channel semantics the sender posts a send descriptor to the QP and the receive descriptor is consumed on the receive queue (RQ) connected to the QP. The sender does not know if the receiver is using an SRQ or a dedicated RQ. Thus, traditionally when using channel semantics the sender has no knowledge or control over the receive buffer.

XRC allows a more flexible form of placement on the receiver. When posting a send descriptor to an XRC QP, the destination *SRQ number* is specified. This allows a sender to specify a different "bucket" for different message sizes as suggested by Shipman, et. al., but without a separate QP.

This same addressing scheme also allows a process to communicate with other processes on the same node as one that it has a QP connection with. Only the SRQ number is needed for addressing, so as long as the SRQ number of the destination is known and at least one XRC QP is connected to a process on the node of the destination, a separate QP is not required.




(a) RC-based Multi-SRQ (RC-MSRQ)

(b) Exclusive XRC-based Multi-SRQ (EXRC-MSRQ)

(c) Shared XRC-based Multi-SRQ (SXRC-MSRQ)

Figure 8.2: Multi-SRQ Designs: Each figure shows two nodes, each with two processes in a fully-connected configuration. Each small white box denotes a QP.

## 8.2 Design

In this section we describe our MPI design using the new XRC transport. We first begin with an overview of the goals in the design, and discuss issues related to the Shared Receive Queues (SRQs) possible connection setup methods.

The main goals of the design are two-fold: First, the design should reduce the memory consumption required for QPs and communication contexts. Second, the design should provide better communication buffer utilization. For example, this means that messages of 900 bytes should only consume a 1KB buffer instead of an 8KB buffer. This means we wish to reduce memory in two ways – both the connection memory as well as the communication buffer memory.

### 8.2.1 Shared Receive Queues

As noted earlier, previous work [70] has shown that communication buffers are not used efficiently when a single pool of receive buffers is used. Instead of using a single pool of

buffers, multiple pools (SRQs) can be allocated. In this section we describe the possible configurations available by using the RC and XRC transports and multiple SRQs.

When using the RC transport this requires a QP for each SRQ available. Figure 8.2(a) shows the connection between two nodes, each with two processes. Despite increasing communication buffer efficiency, this requires a significant amount of connection context memory.

Using the XRC transport and the SRQ addressing scheme, a different QP is no longer required to have this same functionality of selecting a receive buffer pool based on the data size. This allows two different connection models:

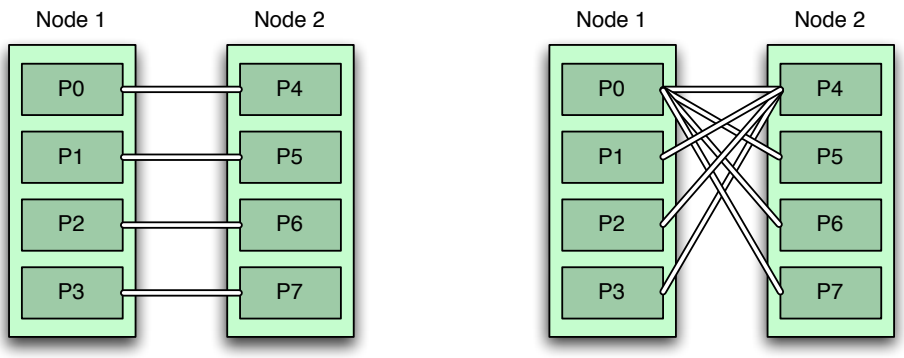
- *Exclusive XRC (EXRC)*: In this model each process still creates a connection (QP) to every other process in the job if needed. There is no use of the XRC ability to connect to processes on the same node with an existing connection. The destination SRQ ability is used to eliminate the additional QPs required in the RC case. This model is shown in Figure 8.2(b).

- *Shared XRC (SXRC)*: Using this model both the additional QPs for multiple SRQs and for processes on the same node are eliminated. This is the method that makes the most of the XRC capabilities. Figure 8.2(c) shows this configuration.

Table 8.1 shows the number of QPs required using these difference schemes. Additional information on best-case and worst-case connection patterns is discussed below.

### 8.2.2 Connection Setup

As mentioned in Section 8.1, XRC allows one connection per node in the optimal case.




(a) Ideal-Case Connection Setup

(b) Worst-Case Connection Setup

Figure 8.3: Depending on communication characteristics, connections may be created differently

To achieve an optimal fully-connected connection pattern each process must have only a single connection to another node. In this chapter, *fully-connected* means that all processes can send and receive from all other processes in the job.

Depending on the connection setup method an ideal setup or worst-case fully-connected pattern may emerge, as shown in Figure 8.3. We isolate two main issues that need to be addressed in a connection setup model:

- *Equal Layout*: Each node must have the same number of processes running on them. In other cases, such as 2 processes on node $A$ and 4 on another node $B$, each of the two processes on node $A$ will require 2 QPs in the best case since each process on $B$ will require one QP to host $A$ in order to send to that host. Clearly, there will be cases where a single connection will then not be possible.

- *Balanced Mapping*: Each process must connect with a peer that has not already created a connection with another process on its same node, otherwise the peer will create two connections to a single node.

We propose three different connection models that are possible for an XRC-based MPI implementation and discuss their advantages and disadvantages:

**Preconnect Method**: If the job will require communication between all peers, connections can be setup at the beginning of the job. This is a static pre-connect method. In this case the optimal connection setup can be made assuming each node has the same number of processes. Even if there are non-equal numbers of nodes, the minimal number of connections can be created. This design has the problem that in many applications many processes do not directly communicate with every other process in the job. Preconnecting the connections can waste a significant amount of memory for a large job.

**Predefined Method**: In this alternative, the connections map is predefined (as in the preconnect method), so the minimal number of connections will be created for each process. The difference between the predefined and preconnect alternatives is that predefined is setup only as needed. The problem with such a design is that it will in many cases require a QP to be setup to a process that it doesn't need to communicate with. This process may also not be expecting any communication either and may be in a computation loop. Unless the connection setup can be done asynchronously, a deadlock could potentially occur.

**On-Demand Method**: In the on-demand method, the minimal connection map is not computed at all. Instead, whenever a process needs to send a message to a process on a node it doesn't have a connection to already, it sends a connect request to the process it needs to

communicate with. In this way a non-minimal connection pattern may emerge. The pattern is dependent on the application.

## 8.3 Experimental Evaluation

In this section we evaluate the designs we described in the previous section. We first start with a description of the experimental platform and methodology. Then we evaluate the memory usage and performance on microbenchmarks and application benchmarks.

### 8.3.1 Experimental Platform

Our experimental test bed is a 64-core ConnectX InfiniBand Linux cluster. Each of the 8 compute nodes has dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of 8 cores per node. Each node has a Mellanox ConnectX DDR HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55], OFED 1.3 release. The proposed designs are integrated into the MVAPICH-Aptus communication device of MVAPICH [53] presented in Chapter 7. We extend the device to allow multiple RC QPs per peer and multiple SRQs. We also extend it to support the XRC transport in both the ESXRC and SXRC modes with any number of SRQs.

All of the designs are implemented into the same code base and the same code flows. As a result, performance differences can be attributed to the transport instead of software differences.

### 8.3.2 Methodology

We evaluate six different combinations:

- *Reliable Connection*: Using the RC transport with a single SRQ (*RC-SRQ*) as well as multiple SRQs (*RC-MSRQ*).

Table 8.1: Comparison of Number of Queue Pairs for Various Channels

| | Attributes | | | QPs per Process | | QPs per Node | |
|---|---|---|---|---|---|---|---|
| | SRQs | Transport | Shared | Best Case | Worst Case | Best Case | Worst Case |
| RC-SRQ | 1 | RC | N | $n \times c$ | $n \times c$ | $n \times c^2$ | $n \times c^2$ |
| RC-MSRQ | 6 | | | $6 \times n \times c$ | $6 \times n \times c$ | $6 \times n \times c^2$ | $6 \times n \times c^2$ |
| EXRC-SRQ | 1 | XRC | N | $n \times c$ | $n \times c$ | $n \times c^2$ | $n \times c^2$ |
| EXRC-MSRQ | 6 | | | | | | |
| SXRC-SRQ | 1 | | Y | $n$ | $n \times c$ | $n \times c$ | $2 \times n \times c$ |
| SXRC-MSRQ | 6 | | | | | | |

- *Exclusive eXtended Reliable Connection*: No sharing connections, but using XRC. Both single SRQ (*EXRC-SRQ*) and multiple SRQs (*EXRC-MSRQ*).

- *Shared eXtended Reliable Connection*: Share connections across nodes. This is using the on-demand connection setup from Section 8.2. Both single (*SXRC-SRQ*) and multiple (*SXRC-MSRQ*) SRQ configurations

Table 8.1 shows a summary of the characteristics of each of these combinations where $n$ is the number of nodes in the job and $c$ is the number of cores per node. We assume for this table that processes are equally distributed. Note that RC-SRQ and EXRC-SRQ are equivalent in the amount of resources required as well as memory efficiency for communication buffers. The EXRC-SRQ case is a control to evaluate whether there are inherent performance differences between the XRC and RC hardware implementations and if the addressing method of XRC adds overhead.

In all of our evaluations we use the "on-demand" connection setup method for XRC. The other connection setup methods will have standard patterns. This method will provide the most insights.

For the multiple SRQ modes, we use 6 SRQs. We use the following sizes: 256 bytes, 512 bytes, 1KB, 2KB, 4KB, and 8KB. Messages above 8KB follow a zero-copy rendezvous protocol.

### 8.3.3 Memory Usage

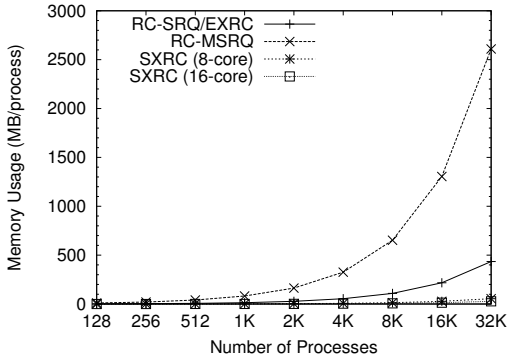
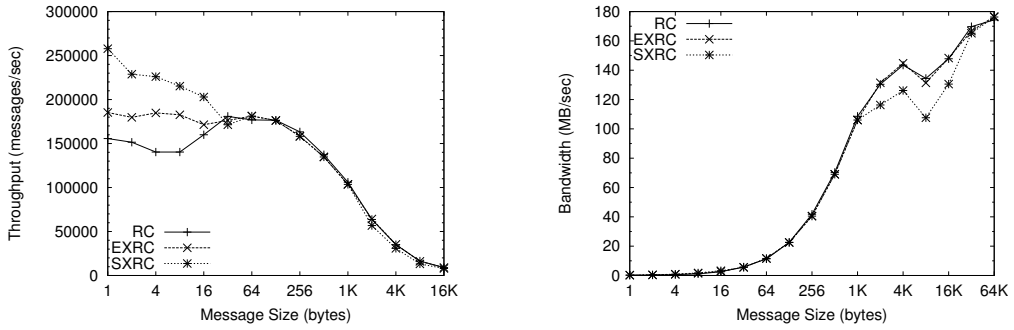



Figure 8.4: Fully-Connected MPI Memory Usage

We first assess the scalability of each of the configuration. Figure 8.4 shows the memory usage when fully-connected. RC-SRQ is the default configuration for most MPIs, one connection per peer process. RC-MSRQ shows the memory usage when 6 SRQs are created per process and therefore the memory usage is six times higher than that of RC-SRQ. The last two lines are the memory usage for the Shared XRC implementations in the best case when in 8-core/node and 16-core/node configurations. EXRC has the same memory footprint as RC-SRQ since a single QP is required still to each process in the job.

From the figure we can observe that a fully-connected job at 32K processes will consume 2.6GB of memory with the RC-MSRQ configuration and 400 MB/process for the




(a) Message Rate

(b) Bandwidth

Figure 8.5: Many-to-Many Benchmark Evaluation

RC-SRQ and EXRC-{M}SRQ configurations. The SXRC designs reduce the memory usage in the best case to 54MB/process and 26MB/process for the 8-core and 16-core configurations, respectively. In the worst case the SXRC design will consume as much as the "RC-Single" model.

### 8.3.4 MPI Microbenchmarks

To assess if there are basic performance differences between the different combinations we ran various standard microbenchmarks. The basic latency, bandwidth, and bi-direction bandwidth results remained very similar across all combinations and are not presented here.

To further assess performance when many peers are being communicated with simultaneously we design a new microbenchmark. In this benchmark each process communicates with a variable number of random peers in the job during each iteration. In this throughput test each process sends and receives a message from 32 randomly selected peers 8 times. We ran this benchmark with 64 processes and report the results in Figure 8.5. From the

figure we can see the SXRC mode is able to achieve higher throughput than the EXRC and RC configurations. In top-end bandwidth the XRC modes are able to outperform the RC mode.

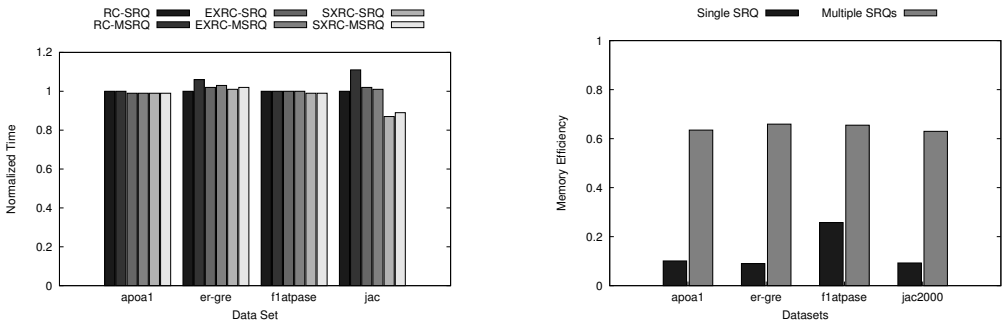
### 8.3.5 Application Benchmarks

In this section we evaluate the configurations against two application-based benchmarks. These are more likely to model real-world use than microbenchmarks. We evaluate using the molecular dynamics application NAMD and the NAS Parallel Benchmarks (NPB). We evaluate all application benchmarks using 64 processes.

#### NAMD

NAMD is a fully-featured, production molecular dynamics program for high performance simulation of large bimolecular systems [59]. NAMD is based on Charm++ parallel objects, which is a machine independent parallel programming system. Of the standard data sets available for use with NAMD, we use the `apoa1`, `f1atpase`, `er-gre`, and `jac2000` datasets.

Figure 8.6(a) shows the overall performance results of the different combinations on the various datasets. From the figure we observe that for both `apoa1` and `f1atpase` performance is very similar across all modes. For `jac` we see that the RC-MSRQ performs 11% worse than the standard RC-SRQ implementation. We believe this is due to HCA cache effects when large numbers of QPs are being used at the same time. By contrast, we see that the SXRC modes provide over 10% improvement. For the same reason as RC-MSRQ performs poorly, the SXRC modes perform well. Since a fewer number of QPs are used they are more likely to stay in the HCA cache. This mirrors what we observed in the many-to-many benchmark in Figure 8.5. We can see in Figure 8.6(b) that communication buffer



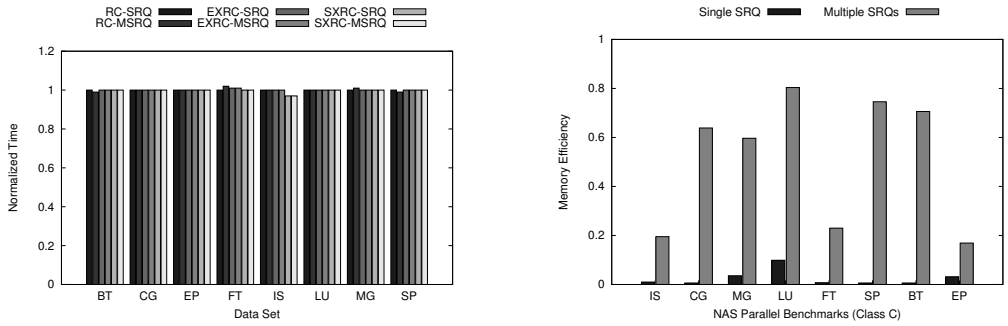

(a) Normalized Time

(b) Memory Efficiency

Figure 8.6: NAMD Evaluation

Table 8.2: NAMD Characteristics Summary (Per Process)

| Benchmark | Configuration | Avg. Comm Peers | Max Comm Peers | Avg. QPs/process | Max QPs/processc | Avg QPs/node | Max QPs/nodee |
|---|---|---|---|---|---|---|---|
| apoa1 | RC-MSRQ | 54.41 | 63 | 285.54 | 336 | 2284.50 | 2340 |
| | RC-SRQ | | | 47.59 | 56 | 380.75 | 390 |
| | EXRC-{M}SRQ | | | 47.59 | 51 | 380.75 | 390 |
| | XRC-{M}SRQ | | | 9.09 | 33 | 72.75 | 80 |
| f1atpase | RC-MSRQ | 62.19 | 63 | 331.50 | 336 | 2652 | 2688 |
| | RC-SRQ | | | 55.25 | 56 | 442 | 448 |
| | EXRC-{M}SRQ | | | 55.25 | 56 | 442 | 448 |
| | XRC-{M}SRQ | | | 9.22 | 33 | 15 | 82 |
| jac2000 | RC-MSRQ | 63 | 63 | 336 | 336 | 2688 | 2688 |
| | RC-SRQ | | | 56 | 56 | 448 | 448 |
| | EXRC-{M}SRQ | | | 56 | 56 | 448 | 448 |
| | XRC-{M}SRQ | | | 9.22 | 33 | 73.75 | 82 |
| er-gre | RC-MSRQ | 25.38 | 63 | 122.28 | 336 | 978 | 1188 |
| | RC-SRQ | | | 19.55 | 56 | 160.25 | 193 |
| | EXRC-{M}SRQ | | | 20.03 | 56 | 160.25 | 194 |
| | XRC-{M}SRQ | | | 8.28 | 33 | 66.25 | 79 |




(a) Normalized Time

(b) Memory Efficiency

Figure 8.7: NAS Parallel Benchmarks (Class C) Evaluation

usage is much improved when using multiple SRQs. This figure shows the ratio of the total amount of received messages to the total amount of memory in the communication buffers used to service those messages.

Table 8.2 shows the connection characteristics of each of the datasets. We can see that each of the datasets requires significant communication, especially `jac` where every process communicates with every other process. For that dataset, we observe for that the sum of the connections for the processes on a single node are only 82 as compared to 448 for RC-SRQ and EXRC modes. The RC-MSRQ configurations require even more QPs, a total of 3136 QPs per node.

**NAS Parallel Benchmarks**

The NAS Parallel Benchmarks [8] are a selection of kernels that are typical in various Computational Fluid Dynamics (CFD) applications. As such, they are a good tool to evaluate the performance of the MPI library and parallel machines. In this evaluation the Class "C" benchmark size was used.

The performance results for each of the configurations are shown in Figure 8.7(a). Very little performance variation was observed in nearly all of the benchmarks. Only one benchmark, IS, showed a consistent improvement with the SXRC transport. The dominating factor in the IS benchmark performance is the `MPI_Alltoall` collective for large message sizes. For large message sizes the `MPI_Alltoall` collective sends directly to each process in the job. In the SXRC configurations, connections can be shared and can reduce connection thrashing in the HCA cache. There seems to be little difference between the SXRC-MSRQ and SXRC-SRQ modes in terms of performance.

Figure 8.7(b) shows the memory efficiency obtained by using multiple SRQs compared to a single SRQ. In all benchmarks efficiency was greatly increased. In the case of SP, efficiency rose from less than 6% to 75%. Using XRC we are able to acheive this buffer efficeincy as well as a reduction in connection memory.

The connection characteristics for the NAS benchmarks are shown in Table 8.3. The benchmarks have a variety of patterns. IS, the benchmark where the SXRC modes outperformed, we notice that all connections are required. Using SXRC each process on average only needs to create 9.25 connections as apposed to 56 connections for the ESRQ and RC-SRQ modes (the shared memory channel is used for 7 others). The on-demand connection setup method seems to work well, although not always setting up the minimal number of connections.

## 8.4 Related Work

Sur, et. al. previously evaluated ConnectX, which is the first HCA to support XRC [77, 47]. Other groups have expressed interest in providing XRC support in MPI, such

Table 8.3: NAS Characteristics Summary (Per Process)

| Benchmark | Configuration | Avg. Comm Peers | Max Comm Peers | Avg. QPs/process | Max QPs/processc | Avg QPs/node | Max QPs/nodee |
|---|---|---|---|---|---|---|---|
| IS | RC-MSRQ | 63 | 63 | 336 | 336 | 2688 | 2688 |
| | RC-SRQ | | | 56 | 56 | 448 | 448 |
| | EXRC-{M}SRQ | | | 56 | 56 | 448 | 448 |
| | XRC-{M}SRQ | | | 9.25 | 25 | 74 | 74 |
| CG | RC-MSRQ | 6.88 | 7 | 23.28 | 24 | 186 | 186 |
| | RC-SRQ | | | 3.88 | 4 | 31 | 31 |
| | EXRC-{M}SRQ | | | 3.88 | 4 | 31 | 31 |
| | XRC-{M}SRQ | | | 3.50 | 4 | 28 | 28 |
| MG | RC-MSRQ | 9 | 9 | 30 | 30 | 240 | 240 |
| | RC-SRQ | | | 5 | 5 | 40 | 40 |
| | EXRC-{M}SRQ | | | 5 | 5 | 40 | 40 |
| | XRC-{M}SRQ | | | 4 | 4 | 32 | 32 |
| LU | RC-MSRQ | 7.50 | 8 | 22.50 | 24 | 180 | 192 |
| | RC-SRQ | | | 3.75 | 4 | 30 | 32 |
| | EXRC-{M}SRQ | | | 3.75 | 4 | 30 | 32 |
| | XRC-{M}SRQ | | | 3.75 | 4 | 30 | 32 |
| FT | RC-MSRQ | 63 | 63 | 336 | 336 | 2688 | 2688 |
| | RC-SRQ | | | 56 | 56 | 448 | 448 |
| | EXRC-{M}SRQ | | | 56 | 56 | 448 | 448 |
| | XRC-{M}SRQ | | | 7 | 7 | 56 | 56 |
| SP | RC-MSRQ | 10 | 10 | 36 | 36 | 288 | 288 |
| | RC-SRQ | | | 6 | 6 | 48 | 48 |
| | EXRC-{M}SRQ | | | 6 | 6 | 48 | 48 |
| | XRC-{M}SRQ | | | 4 | 4 | 32 | 32 |
| BT | RC-MSRQ | 10 | 10 | 36 | 36 | 288 | 288 |
| | RC-SRQ | | | 6 | 6 | 48 | 48 |
| | EXRC-{M}SRQ | | | 6 | 6 | 48 | 48 |
| | XRC-{M}SRQ | | | 4 | 4 | 32 | 32 |
| EP | RC-MSRQ | 6 | 6 | 18 | 18 | 144 | 144 |
| | RC-SRQ | | | 3 | 3 | 24 | 24 |
| | EXRC-{M}SRQ | | | 3 | 3 | 24 | 24 |
| | XRC-{M}SRQ | | | 3 | 3 | 24 | 24 |

as HP MPI and Open MPI [82]. However, there is no detailed study on how this XRC implementation works, the associated design challenges and interactions with applications.

Recently Shipman, et al. in [70] presented a mechanism for efficient utilization of communication buffers using multiple SRQs for different data sizes. For example, a 500 byte message should only consume 512 bytes and a 1.5KB message should only consume 2KB. Current designs have used a single SRQ, where any message will consume a fixed size such as 8KB. This evaluation and design, however, was with the Reliable Connection (RC) transport of InfiniBand, not the XRC transport.

After publication of our work Shipman, et al. proposed X-SRQ [71] for Open MPI, which is similar to our work. It lacks, however, a full analysis of the different methods that are available when using XRC.

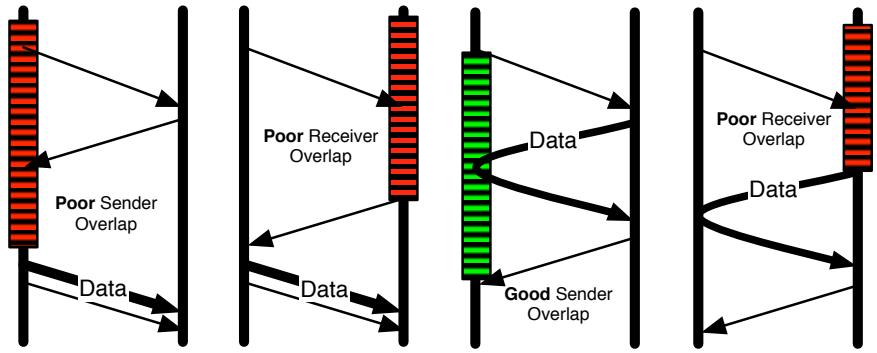
# CHAPTER 9

# INCREASING OVERLAP WITH THE XRC TRANSPORT

MPI provides synchronous and asynchronous data transfer methods, the most basic examples of these being the `MPI_Send` and `MPI_Isend` functions. An `MPI_Isend` does not need to complete when the function returns, instead it can wait for completion at some time in the future. This type of functionality allows for the MPI library to send the data in the background while computation performed. When using an interconnect that provides off-loading of data transfer, meaning the sending and receiving of data can be done without processor involvement, it is even more important that this progress be able to be done in the background. InfiniBand is one such interconnect.

Due to this capability of data transfer offload in many high-performance interconnects, application writers have often tried to restructure their code to use asynchronous MPI calls. Unfortunately for application performance, many MPI libraries do not take advantage of this hardware capability.

In this chapter we explore this problem in depth with InfiniBand and propose *TupleQ*, a novel approach to providing communication and computation overlap in InfiniBand without using threads and instead relying on the network architecture. Instead of using the traditional protocols with RDMA Put and Get operations, we propose using the network hardware to directly place the data into the receive buffers without any control messages or




(a) RDMA Write Protocol (b) RDMA Read Protocol

Figure 9.1: Overlap of Existing RDMA Protocols

intermediate copies. We build on top of the XRC transport described earlier in Chapter 8. We show that our solution is able to provide full overlap with minimal overhead and is able to achieve performance gains of 27% on the SP kernel in the NAS Parallel Benchmarks for 64 processes.

The rest of the chapter is organized as follows: In Section 9.1 we motivate our work by describing the problems with current MPI design for overlap of communication and computation overlap. In Section 9.2, we describe the current implementations of MPI over InfiniBand. TupleQ, our fully-asynchronous design is presented in Section 9.3. We evaluate our prototype on both an overlap benchmark and the NAS Parallel Benchmarks in Section 9.4. We discuss related work in Section 9.5.

## 9.1 Motivation

As noted earlier, MPI allows the application writer to use non-blocking communication with `MPI_Isend` to overlap communication and computation. Unfortunately, the method of

implementing large message transfer, is often done with control messages and an RDMA operation.

Since MPI libraries generally poll for incoming messages, an incoming control message cannot be discovered unless it is in the progress engine (within an MPI call). If the application is trying to achieve overlap of communication with computation, the application will by definition not be in the MPI library. As a result, the control messages can be delayed, leading to no overlap in many cases.

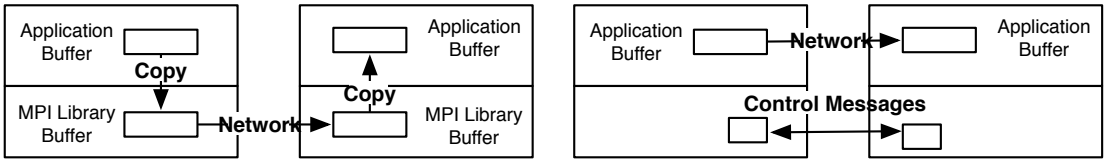
Figure 9.1 shows the overlap that can be achieved with the current designs. The RDMA Write-based design has 3 control messages and leads to poor sender and receiver side overlap. If the sends immediately goes into a computation loop after sending the message there will be no overlap. Similarly, the receiver has no overlap as well. For RDMA Read, the sender has good overlap, however the receiver will have very poor overlap if the receiver has gone into a computational loop. Neither of these existing designs provides good performance.

Others have proposed using threads, however, this increases the overhead since signaled completion in InfiniBand is quite a bit slower than that of polling. This also can decrease performance due to locking required. Signaling has also been suggested to avoid locks [33], however, many calls within the MPI progress engine are not signal-safe.

## 9.2 Existing Designs of MPI over InfiniBand

MPI has been implemented over InfiniBand by many implementors and organizations, however, all of them generally follow the same set of protocols and design:

- *Eager Protocol:* In the eager protocol, the sender task sends the entire message to the receiver without any knowledge of the receiver state. In order to achieve this, the




(a) Eager Protocol (b) Rendezvous Protocol

Figure 9.2: Data Movement of Protocols

receiver must provide sufficient buffers to handle incoming unexpected messages. This protocol has minimal startup overhead and is used to implement low latency message passing for smaller messages.

- *Rendezvous Protocol:* The rendezvous protocol negotiates buffer availability at the receiver side before the message is sent. This protocol is used for transferring large messages, when the sender wishes to verify the receiver has the buffer space to hold the entire message. Using this protocol also easily facilitates zero-copy transfers when the underlying network transport allows for RDMA operations.

### 9.2.1 Eager Protocol

This mode is generally used for small messages and is designed for low-latency and overhead. In this mode the sender pushes the data over to the receiver without contacting the receiver. In this case the sender has no knowledge of the receiver state. This is important for two reasons:

- *Unknown Receive Address*: The address of the application receive buffer is not known. Since the address is not known the RDMA capability of InfiniBand cannot be used for a zero-copy transfer.

- *Unknown Availability*: The sender also has no idea if the receiver has posted a receive for this send operation yet.

As a result of this, the eager protocol for InfiniBand-based MPI libraries is done using copies:

- *Sender Side*: Take a pre-allocated buffer (*send buffer* and place header information (tag, context, and local source rank) at the beginning. Next the application send buffer is copied into the send buffer.

- *Receiver Side*: The message is received into a *receive buffer* by the network hardware. On reception the receiver reads the header and checks the receive queue for a matching receive. If it is found then the data is copied into the corresponding application receive buffer. If it is not available the message is buffered and will be copied when the receive is posted.

This process is shown in Figure 9.2(a). This shows there are two copies as well as the network transfer.

### 9.2.2 Rendezvous Protocol

The rendezvous protocol is generally implemented with one of the RDMA operations of InfiniBand, RDMA Write or RDMA Read.

In each case the sender sends a "Request to Send (RTS)" message to the receiver. Upon receipt of the RTS message the queue of posted receives is searched. Then depending on the protocol different steps are taken:

- *RDMA Write (RPUT)*: If the receive for this send has already been posted the address of the application receive buffer is sent back to the sending process. If the receive has

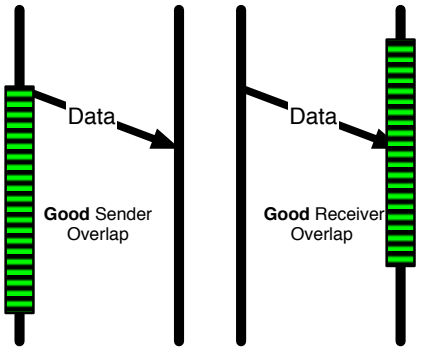
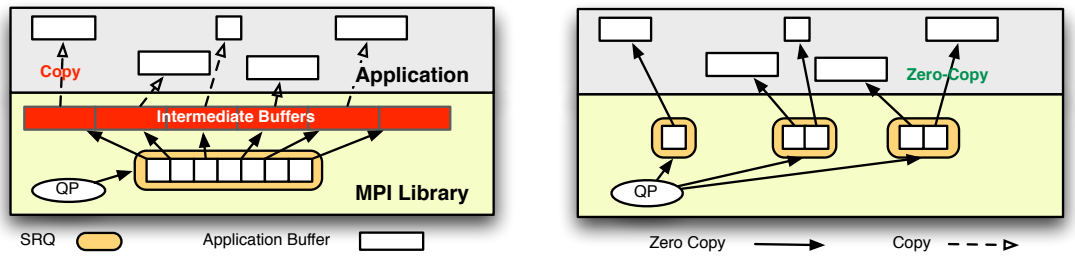



Figure 9.3: TupleQ Overlap

not been posted the address is sent whenever the receive is posted. Upon receipt of the receiver buffer, the sender directly RDMA writes the data from the sender application buffer to the receiver application buffer on the remote node. A finish message (FIN) is also sent to the receiver to notify the completion of the send peration.

- *RDMA Read (RGET)*: If the receive has already been posted, the receiver directly reads the data from the sender's application buffer and places it into the receive buffer with an RDMA Read operation. If the receive has not been posted then the operation will occur after the receive has been posted. After completion of the RDMA Read the receiver sends a FIN to the sender to indicate that the send can be marked complete.

In both of these cases the application buffer undergoes no intermediate copies, "zero-copy transfer," but does require at least two control messages to be sent. This data movement path is shown in Figure 9.2(b). As we will show in the next section, these control messages often prevent communication and computation overlap.




(a) Traditional Copy-based (b) TupleQ Zero-Copy

Figure 9.4: Transfer Mechanism Comparison

## 9.3 Proposed Design

In this section we describe our design that allows full overlap of communication and computation for all message sizes. We first describe the mechanism that we use to provide zero-copy transfers without needing to exchange control messages, then we discuss the option to use sender-side copies, creation of receive queues, and discuss handling of the MPI wildcards.

### 9.3.1 Providing Full Overlap

As noted in Section 2.1, InfiniBand offers two sets of semantics to transfer data, channel and memory. Traditionally memory semantics (RDMA) has been seen as the only efficient method by which to transfer large messages without copies for MPI. Our design shows that this is not the case.

In channel semantics the receive buffers are consumed in order from a receive queue. In contrast, MPI semantics require messages to be matched in order, not necessarily in the same order that they are posted. This semantic gap has been seen as a need for exchanging control messages.

After studying various applications and patterns we have found that relatively few tags and communicators are used in MPI libraries. As such, we propose to have *separate queues for each matching tuple*. The matching tuple contains the tag, communicator ID and rank. If each matching tuple has a separate queue, MPI semantics can match those of InfiniBand as long as only messages with that tuple are sent to that receive queue. In this way we are able to provide full overlap as seen in Figure 9.3.

In addition to being zero-copy without control messages, this is fully asynchronous and receives are now "matched" by the hardware. If a receive buffer is not posted to a queue and a send operation is sent to that queue, the sender HCA will block until the receive buffer is posted. In this case the HCA is handling all operations and neither the sender or receiver CPU has any involvement with these retries.

### 9.3.2 Creating Receive Queues

The RC transport of InfiniBand only allows a Queue Pair (QP) to be connected to a single SRQ or RQ. As a result, a new connection would be required for each new queue. This approach would not be scalable. In contrast, the XRC transport allows addressing of receive queues, so multiple QPs are not required. Instead we just create a new SRQ for each matching tuple. When sending a message, the sender simply sends to the previously agreed upon SRQ number of the matching tuple.

Given the very large space of possible matching tuples, the receive queues are created "on demand." Only when the sender needs to send a message of a given tuple is an out-of-band message sent to the receiver to setup a queue for that tuple. The receiver responds with the SRQ number for the tuple. This value is then cached on the sender, so all future sends of that tuple will use that queue. Similarly, the receiver will post all application

receive buffers directly to the receive queue for that tuple. If the tuple has not been created when the receive queue will be created for that tuple.

### 9.3.3 Sender-Side Copy

In MPI buffer availability determines when a blocking send call (`MPI_Send`) call can complete. So as long as the send buffer is available to be overwritten, the call can complete. As a result, as described earlier, since many MPI implementations already copy the send buffer to another buffer the send can be marked complete directly after the buffer has been copied. This option is also possible in TupleQ as well since there may be benefits to allowing the sender to go on even if the receiver has not posted the receive. We will evaluate this option.

### 9.3.4 MPI Wildcards

MPI specifies two wildcards that may be used when posting a receive. `MPI_ANY_SOURCE` and `MPI_ANY_TAG`. The first, `MPI_ANY_SOURCE`, means that the source of the message is unknown and can match any source. The second, `MPI_ANY_TAG`, allows a receive operation to match any tag. Both can be used together to match any incoming send operation.

These wildcards are a challenge for for a number of reasons:

- There is no "wildcard" receive that can be placed in many receive queues at a time and then removed from all when it is consumed
- A sender will try to always send to the designated tuple queue and the hardware will not complete the message until a receive is posted. The receiver will not know what buffer to post the receive to since it has been given a wildcard.

To address this issue we introduce a wildcard fallback mechanism. When a wildcard receive is posted all connections to the receiver are shutdown. This prevents any messages to be retried by the HCA and all will end up connecting back to the receiver with a traditional eager/rendezvous mechanisms described in Section 2.2. After the wildcard has been matched the receiver will notify the senders and the tuple queues can be used again.

This fallback can be quite expensive if it occurs too often, so TupleQ will fall back to a traditional implementation model if it occurs too frequently. Alternatively, it can be disabled for applications that are known to contain wildcards. Further, the MPI-3 standard that is under discussion may contain support for "asserts", in which the application could inform the MPI library that it will or will not be using wildcards.

To provide full functionality we would like to see a hardware matching mechanism that allows a single receive descriptor to be posted to multiple queues simultaneously and be removed from all queues once it is consumed.

## 9.4 Evaluation

In this section we evaluate the design described in the previous section. We first measure the overlap potential of our design as well as the overhead incurred as compared to the traditional implementation. We also evaluate our design with the SP kernel of the NAS Parallel Benchmarks.

### 9.4.1 Experimental Platform

Our experimental platform is a 128-core InfiniBand Linux cluster. Each of the 8 compute nodes has 4 sockets each with a Quad-Core AMD Opteron 8350 2GHz Processor with 512 KB L2 cache and 2 MB L3 cache per core. Each node has a Mellanox MT25418

dual-port ConnectX HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55], OFED 1.3 release.

### 9.4.2 Experimental Combinations

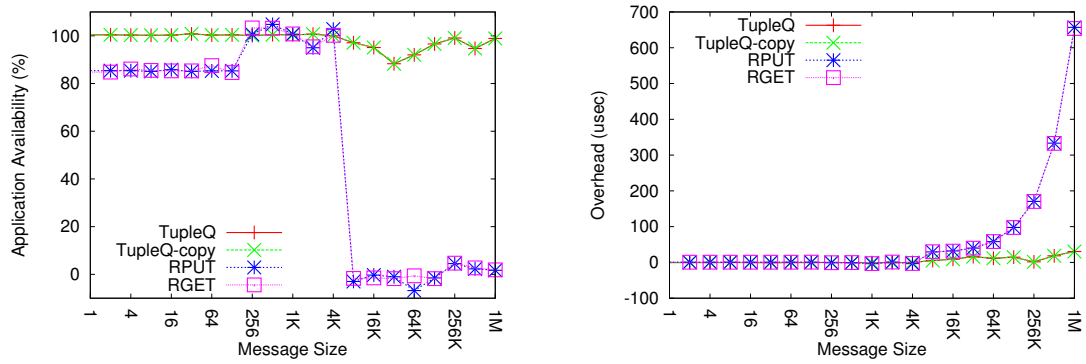
We evaluate four different combinations to observe their effect on overlap and performance. For the MPI library, we use MVAPICH [53], a popular open-source MPI implementation over InfiniBand. It is based on MPICH [24] and MVICH [34] and is used by over 760 organizations worldwide. We implement our TupleQ design into MVAPICH as well.

The combinations we evaluate are:

- *TupleQ*: This is the design described in Section 9.3 with no data copies.
- *TupleQ-copy*: This is the design described in Section 9.3 with sender-side copies for messages under 8KB.
- *RPUT*: This is the RDMA Write based design from MVAPICH
- *RGET*: This is the RDMA Get based design from MVAPICH

### 9.4.3 Overlap

We use the Sandia Benchmark [66] to evaluate the overlap performance of our design. Figure 9.5(a) shows the Application Availability that the protocol allows. Due to the control messages, the RGET and RPUT designs have poor overlap. 8KB is the threshold where the rendezvous protocol is used for RPUT and RGET and thus the steep drop in overlap. Since the TupleQ design is fully asynchronous nearly full overlap is obtained for all message sizes, including small ones. Figure 9.5(b) shows the overhead incurred with each send




(a) Application Availability

(b) Communication Overhead

Figure 9.5: Sandia Overlap Benchmark

operation – what is not overlapped. Since the TupleQ design does full overlap there is minimal overhead, whereas the RGET and RPUT designs have high overhead since none of the message transfer can be overlapped.

### 9.4.4 NAS Parallel Benchmarks (NPB) - SP

The NAS Parallel Benchmarks [8] (NPB) 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.

Of these kernels, the SP kernel attempts to provide overlap of communication and computation overlap with `MPI_Isend` operations. As such, this is the kernel that we evaluate for the performance of our new design.

We evaluate this benchmark using 64 processes. Further, we disable shared memory for all combinations since the shared memory implementation is not overlapping (future

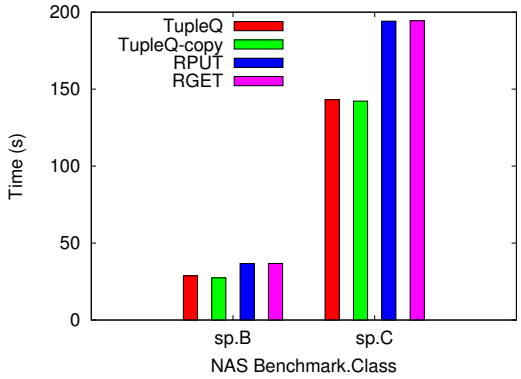



Figure 9.6: NAS SP Benchmark

designs have been proposed that do allow overlap) and will negate some of the benefit seen.

As shown in Figure 9.6, the new TupleQ design does very well for both Class B and Class C. For Class B, the TupleQ design gives 27.38 seconds and the copy design gives 28.74 seconds. The RGET and RPUT designs do similar with 36.78s and 36.69, respectively. The same pattern is shown for Class C. The TupleQ design is able to give 27% higher performance. Again the RPUT and RGET designs perform similarly. There is very little difference between the TupleQ and TupleQ-copy modes. There are smaller messages being transfered, but the majority of the transfers are large with these datasets. Given that the TupleQ design requires no buffering it is the better option.

## 9.5 Related Work

Achieving good overlap between computation and communication in MPI libraries has been a hot topic of research. Brightwell et al. [14] have demonstrated the application

benefit from good overlap. Eicken et al. [88] have proposed hardware mechanisms to provide better overlap between computation and communication.

In terms of InfiniBand, Surs et al. [80] proposed a RDMA read based rendezvous protocol with a separate communication thread to achieve overlap. Kumar et al. [33] have proposed a lock free variant of the RDMA based design based on signals and a thread.

Our work is different as we do not use any threads to perform progress, as both of the previous designs have done. As a result we do not require any locking, signals, or other library interaction. Additionally, we do not use RDMA operations, which all other MPI libraries over InfiniBand use to implement large message transfer.

# CHAPTER 10

# SCALABLE AND EFFICIENT RDMA SMALL MESSAGE TRANSFER

To obtain the lowest latency, MPI library implementations over InfiniBand generally include support for small message transfer using Remote Data Memory Access (RDMA) Write operations [43]. This transfer mode is referred to as "RC-FP," "RDMA Fast Path" or "Eager RDMA," depending on the developer, however, they all follow the same implementation and design. This basic design is used in other libraries other than MPI including its usage in some GASNet [11] implementations.

Although this "Fast Path" design has been shown to improve latency, it requires a large amount of memory to be used within the implementing library. This is because the design uses fixed-sized persistent buffers. This means that the sender and the receiver must both have dedicated memory for each other. This typically means 256 KB of memory is required for both the sender and receiver. For bi-directional communication using RDMA fast path an additional 256 KB is required for each side for a total of 1 MB per bi-directional connection. For a large number of channels, the memory usage can be significant.

In this chapter we propose a new design, *Veloblock*[3], for message transfer using RDMA Write operations. This novel design eliminates the need for persistent fixed-size buffers.

[3] 'Velo-' is the Latin root for 'fast', and 'block' refers to the view of memory in the design.

Messages only take up as much memory as they require and the sender side no longer needs to have a set of dedicated buffers. Instead of small 32 byte message taking up a full 8 KB buffer, it can now only consume 32 bytes. This can significantly reduce the memory by a factor of 16 times from 512 KB per pair to 32 KB. We show that our design is able to outperform a non-RDMA fast path enabled design by 13% for the AMG2006 multigrid physics solver. We also outperform a traditional RDMA fast path design by 3%, while using 16 times less memory.

The remaining parts of the chapter are organized as follows: Section 10.1 presents the existing RDMA fast path design. New design options for RDMA fast path and our proposed *Veloblock* design is presented in Section 10.2. Evaluation and analysis of an implementation of our design is covered in Section 10.3. Section 10.4 discusses work related to our own.

## 10.1 RDMA Fast Path and Existing Design

In this section we describe the existing designs for RDMA Fast Path. The first RDMA Fast Path design was described in 2003 [43] and has remained mostly unchanged since then. We first give a brief overview of what "RDMA Fast Path" means and how current designs have been implemented.

### 10.1.1 What is RDMA Fast Path?

In general RDMA fast path refers to a message passing mode where small messages are transferred using RDMA Write operations. Instead of waiting for completion queue (CQ) message, the receiver continually polls a memory location waiting for a byte change.

Waiting for a byte change leads to lower latency than waiting for a completion queue entry. Using any sort of notification negates the performance benefit. When this mode

was originally proposed the difference in latency on the first-generation InfiniBand cards between RDMA fast path and the normal channel semantics of InfiniBand was 6$\mu$sec to 7.5$\mu$sec [43]. The difference on our fourth-generation ConnectX InfiniBand is much lower, but there is still a processing overhead for the card to signal receive completion.

This mode can be achieved since some adapters, such as all Mellanox InfiniBand HCAs, guarantee that messages will be written in order to the destination buffer. The last byte will always be written last.

### 10.1.2 Existing Structure

The structure of existing RDMA fast path designs is to have fixed-size chunks within a large buffer. Figure 10.1 shows this structure.

On the sending side, the sender selects the next available send buffer and copies the message into the buffer and performs an RDMA Write operation to the corresponding buffer on the receiver. The receiver polls on the next buffer where it is expecting a message. Upon detecting the byte change it can process the message. It can send either an explicit message to the sender to notify it that the buffer is available again or piggyback that information within message headers.

### 10.1.3 Detecting Message Arrival

To detect the byte change the receiver must set the buffer into a known state prior to any data being allowed to be placed into it. Recall that to use this mode the hardware must write data in order, so the last byte must be changed to know that the entire message has been received. This traditional mode uses a head and tail flag mode. As seen in Figure 10.2, the head flag is first detected. If the head flag has been changed, then see if the tail flag at $base\_address + size$ is equal to the head value. If it is equal, the data has arrived. To

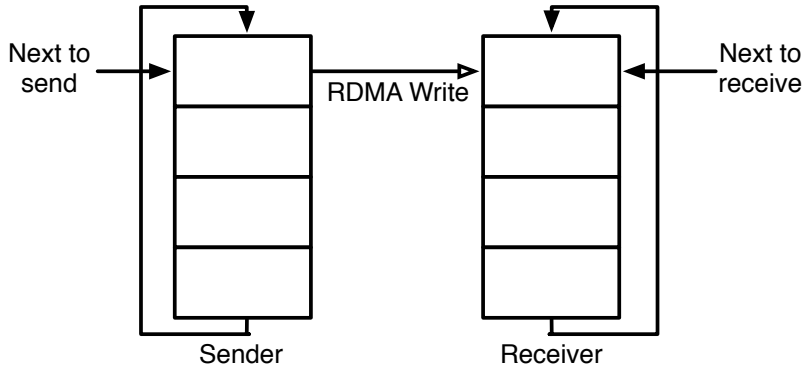



Figure 10.1: Basic structure of paired buffers on sender and receiver in the RDMA fast path design

ensure that the tail flag differs from the previous value at that address the sender *must keep a copy of the data* that it previously sent. Thus the sending side must have the same amount of buffer reserved as the receiver for this channel. Since the message is filled from the beginning or "top" of the buffer we refer to this design as "top-fill."

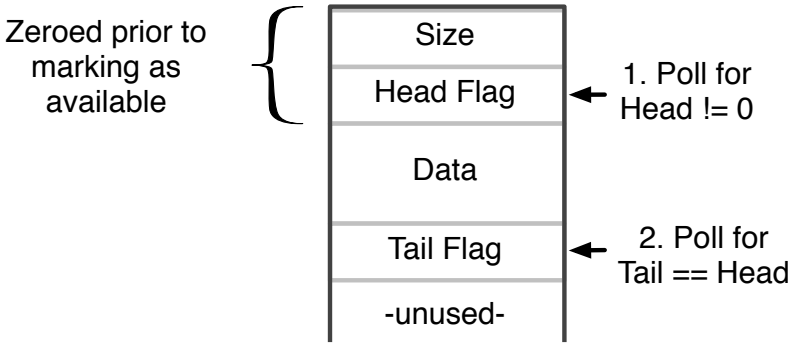



Figure 10.2: Message Detection Method for fixed-length segments

## 10.2 Veloblock Design

In this section we describe the design issues for our new RDMA fast path design, *Veloblock*. First we describe the goals of the design, followed by our solutions to achieve these goals, the various options available, and then finally the design that we select.

The broad goal for a new RDMA Fast Path design is to retain the benefits of lower latency and overhead by using RDMA Write operations for small messages, but reduce the amount of memory required. Memory usage for the RDMA fast path for each process comes from:

$$N_{buffers} \times B_{size} \times (P_{send} + P_{recv})$$

Where $P_{send}$ and $P_{recv}$ are the number of send and receive fast path peers, $N_{buffers}$ is the number of buffers and $B_{size}$ is the size of each buffer. While simply reducing $P$, $N$ or $B$ can save memory, it can also reduce the performance.

Our approach to reduce the memory usage is two-fold:

- Remove the sender-side buffer. This will reduce the amount of memory required by half.
- Use memory as a "block" instead of as fixed 8 KB segments.

### 10.2.1 Remove Sender-Side Buffer

The sender-side buffer is a requirement due to the tail flag in the existing design. The head/tail flag must be selected to be something other than the current value at the tail buffer location. If the value at that position the buffer was already set to the head value it would incorrectly think the message had arrived resulting in invalid data.

To remove the sender-side buffer we propose using a "bottom-fill" approach. Using such an approach there is only a need for a tail flag instead of both head and tail flags. Addionally, there is no need to know the previous value of the tail byte – this byte can just be zeroed out at the receiver before a message arrives. This design can be seen in Figure 10.4(b). Note, an approach of doing a `memset` of zeros on the entire receiver buffer could also remove the need for a sender-side buffer in the top-fill design, but this also incurs a prohibitively large overhead.

### 10.2.2 Supporting Variable-Length Segments

To the best of our knowledge, all current designs of RDMA fast path use fixed-size buffers. In general these messages blocks are 8 KB or larger. However, as mentioned earlier, using fixed width buffers can be very inefficient. Clearly a variable width buffer can increase memory efficiency since a 32 byte message now only consumes 32 bytes rather than an entire 8 KB chunk.

#### Detecting Arrival

Note that a decision here has an impact on how a message can be detected. If a message no longer arrives at a pre-established location the next arrival bit will still need to be changed to zero via some method.

In existing designs the arrival bit can be zeroed out since the next arrival location is always known. Without zeroing out an entire buffer, it is not possible to always zero out the next byte on the receiver side without an additional operation.

To achieve the zeroing of the next arrival byte we propose sending an extra zero byte at either the beginning or the end of the message. Figure 10.3 shows how an extra byte can be sent in the variable bottom-fill design. In a top-fill design the extra zero byte is sent at

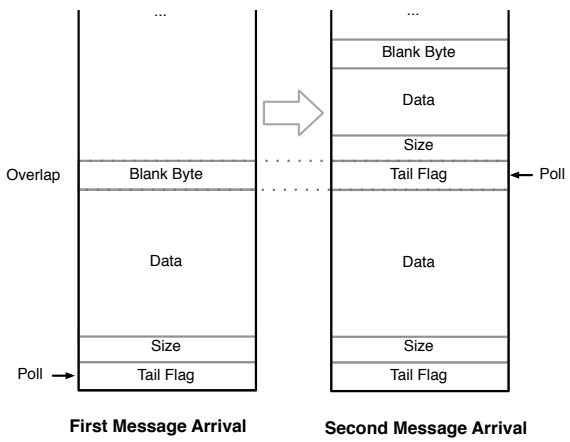



Figure 10.3: Message Detection Method for variable-length segments

the end of the message instead of the beginning. By sending this extra byte of data we are able to reset the bit where the next section of data is to arrive.

### Flow Control

When using fixed-width segments flow control is generally 'credit-based,' where each buffer is a credit. So after the receiver consumes the message and the receive buffer is free the receiver can signal the sender that the buffer is available again. When using the remote memory as a block (variable length) instead of fixed-segments the credit is now based on bytes. When a receiver processes a message it can tell the sender the number of additional bytes in the buffer that are now available. This can be done as a piggyback in message headers, or an explicit message. This type of notification is similar to credit control in existing designs.

Table 10.1: Comparison of RDMA Fast Path Designs

| | Characteristics | | Memory Usage | |
|---|---|---|---|---|
| | Top/Bottom Fill | Variable/Fixed | Sender Buffer | Efficiency |
| Fixed Top-Fill | Top Fill | Fixed | Required | Low |
| Fixed Bottom-Fill | Bottom Fill | Fixed | Not Required | Low |
| Variable Top-Fill | Top Fill | Variable | Required | High |
| Variable Bottom-Fill | Bottom Fill | Variable | Not Required | High |

### 10.2.3 Possible Designs

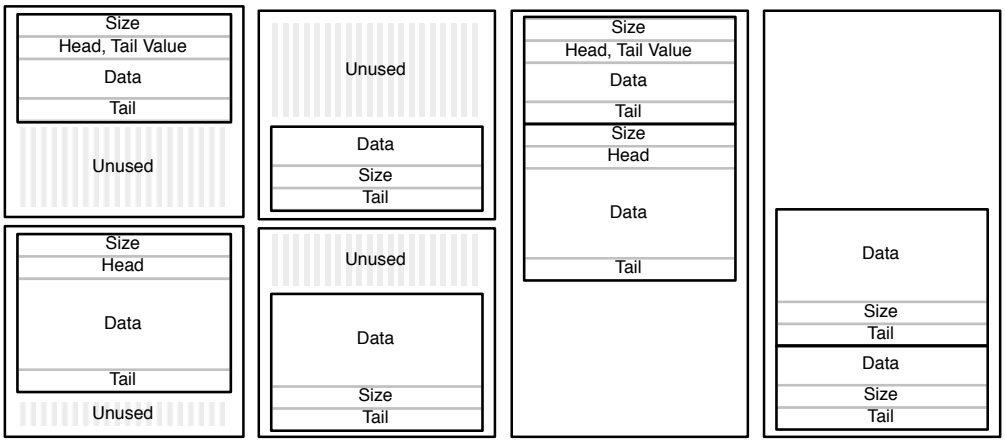
Using these parameters there are four possible design options that can be created with these parameters. Each of these options is shown in Figure 10.4. Table 10.1 shows the features of each design.

- *Fixed Top-fill*: In this design fixed buffers are used and filled from the top. This design requires buffers on the sender and receiver to be dedicated to each other. This is the most memory inefficient design. This is the design employed by MVAPICH [53], Open MPI [82] and others.

- *Fixed Bottom-fill*: This method uses fixed buffers, however, unlike the top-fill design it does not require dedicated buffers on the send side.

- *Variable Top-fill*: In this mode only the required amount of buffer space is used, however, it does require a dedicated sender-side buffer.

- *Variable Bottom-fill / Veloblock*: This mode is the most memory efficient. Messages only take as much memory as required and does not require a dedicated sender-side buffer.




(a) Fixed Top-fill (Original)

(b) Fixed Bottom-fill

(c) Variable Top-fill

(d) Variable Bottom-fill (Veloblock)

Figure 10.4: Fast Path Designs: Each figure shows one of the options available for designing a RDMA fast path. Note that all "top-fill" designs also need a mirrored sender-side buffer.

### 10.2.4 Veloblock Design

Given these options, the highest memory efficiency will come from using the *Variable Bottom-fill* method. This is the method that we propose using and give the name *Veloblock*. Using this method messages only take as much room as they need rather than an entire block.

With this design the memory requirements can be described as the following:

$$B_{nsize} \times P_{recv}$$

Note that $P_{send}$ is eliminated from this equation. Here only the block size and number of peers that a process is receiving from are involved. $B_{nsize}$ here is larger than that of the

original case ($B_{size}$), but since small messages only take up as much space as required it can be significantly less than $N_{buffers} \times B_{size}$ of the original case.

The basic MVAPICH implementation allocates 32 buffers, each of size 8 KB for each connection. This means that a receiver must allocate $32 \times 8$ KB $= 256$ KB of memory and since it is a top-fill design 256 KB on the sender side as well for a total of 512 KB. With the variable bottom-fill Veloblock design we can instead allocate one larger buffer and have messages flow in as needed. In the next section we will run Veloblock with only a 32 KB buffer and observe the performance.

## 10.3 Veloblock Evaluation

In this section we evaluate the design we proposed in the previous section. We first start with a description of the experimental platform and methodology. Then we evaluate the memory usage and performance on microbenchmarks and application benchmarks.

### 10.3.1 Experimental Platform

Our experimental test bed is a 128-core ConnectX InfiniBand Linux cluster. Each of the 8 compute nodes a quad socket, quad core AMD "Barcelona" processors for a total of 16 cores per node. Each node has a Mellanox ConnectX DDR HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55], OFED 1.3 release. The proposed designs are integrated into the MVAPICH-Hybrid communication device of MVAPICH [53] explored in Chapter 7.

All of the designs are implemented into the same code base and the same code flows. As a result, performance differences can be attributed to our design instead of software differences. All experiments are run using the Reliable Connection (RC) transport of InfiniBand.

### 10.3.2 Methodology

We evaluate three different combinations:

- *Original*: The existing design described in Section 10.2 as Fixed Top-fill. This design consumes 512 KB of memory total per uni-directional channel (256KB on the receiver and 256KB on the sender).

- *Original-Reduced*: This is the same design as *Original*, however, we restrict the amount of memory to 32 KB to match that of our new design. This will show if our proposed design is necessary or if buffers could simply be reduced in the basic design.

- *Veloblock*: This is our new design proposed in Section 10.2 that uses memory as a block instead of chunks of pre-determined size. This uses 32 KB of memory total per channel.

We also want to observe the effect of the number of fast path channels on application performance. In particular, we want to see if additional channels can increase performance. Due to the memory usage many implementations limit the number of fast path channels allowed – Open MPI for example allows only 8 by default. We will also track the amount of memory required for fast path communication.

### 10.3.3 Application Benchmarks

In this section we evaluate each of our configurations using two application benchmarks: AMG2006 and LAMMPS.

Table 10.2: Communication Characteristics

| Application | Dataset | Remote Peers | Configuration | Fast Path Memory (per process) | Remote Messages over Fast Path (%) |
|---|---|---|---|---|---|
| AMG2006 | Refinement 3 | 103.92 | Original | 51.95 MB | 89.75 |
| | | | Original-Reduced | 3.25 MB | 49.54 |
| | | | Veloblock | 3.25 MB | 87.22 |
| | Refinement 4 | 107.16 | Original | 53.58 MB | 90.25 |
| | | | Original-Reduced | 3.35 MB | 43.49 |
| | | | Veloblock | 3.35 MB | 89.23 |
| | Refinement 5 | 109.12 | Original | 54.56 MB | 89.99 |
| | | | Original-Reduced | 3.41 MB | 49.38 |
| | | | Veloblock | 3.41 MB | 88.36 |
| NAMD | jac | 59.13 | Original | 29.57 MB | 83.51 |
| | | | Original-Reduced | 1.85 MB | 52.79 |
| | | | Veloblock | 1.85 MB | 83.53 |
| | ergre | 18.88 | Original | 9.44 MB | 64.69 |
| | | | Original-Reduced | 0.59 MB | 56.71 |
| | | | Veloblock | 0.59 MB | 64.75 |
| | apoa1 | 90.16 | Original | 45.08 MB | 72.51 |
| | | | Original-Reduced | 2.81 MB | 58.26 |
| | | | Veloblock | 2.81 MB | 72.52 |

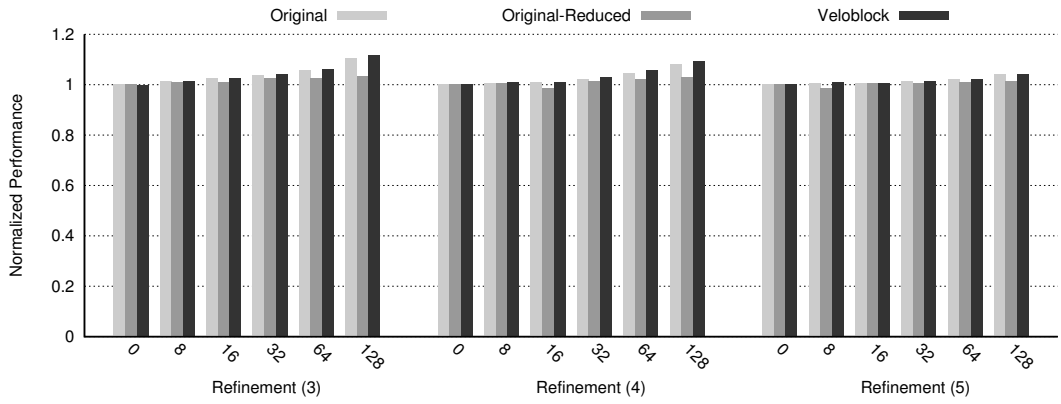



Figure 10.5: AMG2006 Performance (higher bars are better). The "0, 8, . . . , 128" values refer to the number of RDMA fast path connections allowed to be created per process.

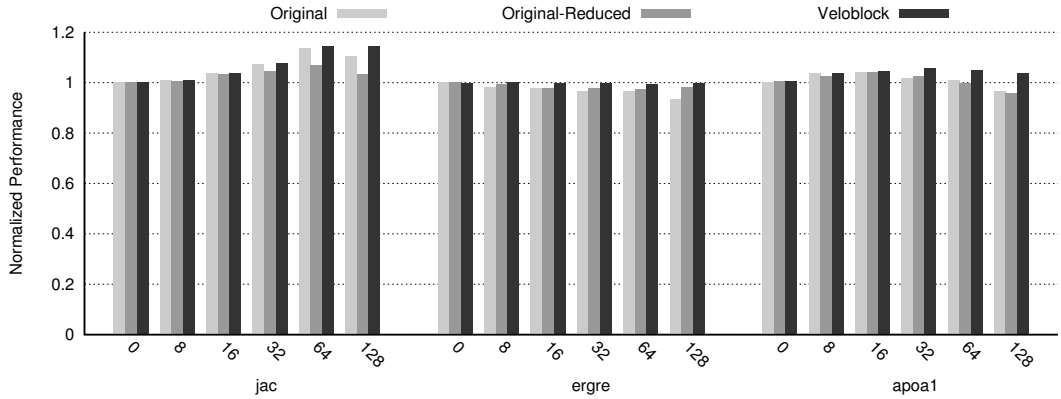



Figure 10.6: NAMD Performance (higher bars are better)

### AMG2006

AMG2006 is a parallel algebraic multigrid solver for unstructured meshes. It is part of the ASC Benchmark Suite [5]. The benchmark and results we show are given using the default driver for the benchmark. It is very communication intensive and communication time can be a significant portion of the overall performance [5].

Figure 10.5 shows the performance of AMG2006 with varied refinement levels. Lower refinement values result in higher degrees of communication. We can see from the figure that using RDMA fast path channels clearly increases performance. For a refinement value of 3 an increase in performance of 13% for 128 *Veloblock* channels over the configuration where no RDMA fast path channels are created is observed. We also note that the *Original-Reduced* mode with 128 channels performs only 3% higher than the base case with no channels created. Additionally, the *Veloblock* design outperforms the *Original* design by 3%, while using significantly less memory.

Table 10.2 shows the characteristics of each of the configurations. For the refinement value of 3, the number of remote peers is on average 103.92. This average is fractional since some processes have more peers than others. When allowing up to 128 fast path connections to be created, the *Original* design will consume 52 MB of memory per process. By contrast, our new design will use only 3.25 MB/process. Note that fast path connections are created "on demand," so only if a process is communicating with a remote process will a fast path buffer be allocated.

In Table 10.2 we also present the percentage of remote messages that are able to use the RDMA fast path channel. If there is not an available remote buffer the sender cannot use RDMA fast path. Thus, for the *Original-Reduced* configuration which has less buffers we note that only 50% of remote messages can use the RDMA fast path, where as for *Original* and *Veloblock* nearly 90% take the fast path. Note that some messages are larger than 8KB and cannot take the fast path, so 100% is often not possible.

**NAMD**

NAMD is a fully-featured, production molecular dynamics program for high performance simulation of large bimolecular systems [59]. NAMD is based on Charm++ parallel objects, which is a machine independent parallel programming system. It is known to scale to thousands of processors on high-end parallel systems. Of the standard data sets available for use with NAMD, we use the `apoa1`, `er-gre`, and `jac2000` datasets. We evaluate all data sets with 128 tasks.

Figure 10.6 shows the performance of each of these three datasets with increasing numbers of allowed fast path connections. The first of the benchmarks, `jac`, is the Joint-Amber Charmm Benchmark and is very communication intensive. For this benchmark the *Veloblock* design again performs the best with a 14% improvement over the case with no

RDMA fast path. The original configuration is a 12% improvement, but requires a significant amount of memory. The *Original-Reduced* configuration, which uses the same amount of memory as our new design, shows only a 4% improvement.

From Table 10.2 we can see that for the `jac` benchmark nearly 84% of messages are able to use the fast path for both *Original* and *Veloblock* and only 53% for *Original-Reduced*. This explains the performance gap in the results. The new *Veloblock* design additionally consumes 42 MB of memory less per process than the *Original* mode.

For the `apoa1` benchmark the *Veloblock* continues to show benefits with up to a 6% improvement over the base case with no RDMA fast path connections. We do notice that after 32 connections performance gets slightly worse. We attribute this to the additional time to poll for message arrival when additional connections are added. If not enough messages are transferred over this path then the polling overhead can exceed the benefit. For example, with `apoa1`, increasing from 32 fast path connections to 64 the number of messages taking the fast path only increases from 36.40% to 47.78%.

The `ergre` benchmark shows a similar trend where the polling for the *Original* designs seems to lower performance. To avoid these types of problems RDMA fast path implementations can tear down connections that do not meet a pre-configured message rate.

## 10.4 Related Work

Many other works have proposed methods of increasing the efficiency of communication buffers. The work most similiar to ours is Portals [13], in which buffers are allocated from a block as they arrive, so space is the limiting factor rather than the number of messages. This work, however, was not done with the InfiniBand and had additional NIC

features to allow this flow. This style of support was also suggested for VIA, a predecessor of InfiniBand, but was never adopted into hardware [12].

Balaji also explored the idea of a block-based flow control for InfiniBand for the Sockets Direct Protocol (SDP) [9]. In this work RDMA Write with Immediate operations were used, which eliminates the benefit of lower latency. It also does not have to address the issues to clearing out the arrival bytes since it uses the Completion Queue (CQ) entry.

# CHAPTER 11

# PROVIDING EXTENDED RELIABILITY SEMANTICS

Clusters featuring the InfiniBand interconnect are continuing to scale. As an example, the "Ranger" system at the Texas Advanced Computing Center (TACC) includes over 60,000 cores with nearly 4000 InfiniBand ports [81]. The latest list shows over 28% of systems and over 50% of the top 100 are now using InfiniBand as the compute node interconnect.

Along with this increasing scale of commodity-based clusters is a concern with the the Mean Time Between Failure (MTBF). As clusters continue to scale higher, it becomes inevitable that some failure will occur in the system and hardware and software need to be designed to recover from these failures. One very important aspect of providing reliability is to guard against any network failures.

In this chapter we design a light-weight reliability protocol for message passing software. As an example, we use the InfiniBand interconnect given its prominence in many HPC deployments. We explain our design that leverages network features to provide a high-performance yet resilient design. We refer to resiliency as being able to recover from network failures including core switches and cables, but also failures on the end-node network devices themselves. We propose designs for both the eager and rendezvous protocols

in MPI and support for both Remote Direct Memory Access (RDMA) read and put operations. We show that our proposed method has an insignificant overhead and is able to recover from many categories of network failures that may occur. We show that our design also uses limited memory for additional buffering of messages.

This chapter is organized as follows: Section 11.1 describes the reliability semantics and error notification of InfiniBand. Our reliability design is presented in Section 11.2. Section 11.3 gives our experimental setup and Section 11.4 is an evaluation and analysis of an implementation of our design. Work related to our design is presented in Section 11.5.

## 11.1 InfiniBand States and Reliability Semantics

In this section we first describe the Queue Pair (QP) states followed by a discussion on the reliability semantics of InfiniBand. Lastly, we describe how error states are communicated from the hardware.

### 11.1.1 Queue Pair States

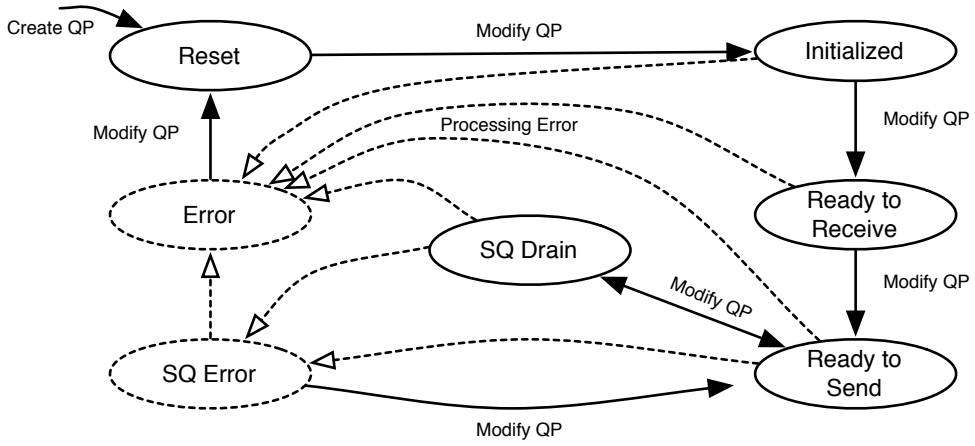



Figure 11.1: Queue Pair State Diagram

Queue Pairs (QPs) are communication endpoints. In the case of RC, a single QP is a connection to another QP in another process. Each QP has its own state, each with different characteristics. Figure 11.1 shows the state diagram for each of these states:

- *Reset*: This is base state of a QP after creation. No work requests can be processed.
- *Init*: After moving from Reset, new receive requests can be issues, but no incoming or outgoing requests are satisfied.
- *Ready to Receive (RTR)*: In this mode a QP can process receive requests, but will not process any outgoing send requests.
- *Ready to Send (RTS)*: This is the working state of the QP. Incoming and outgoing requests are serviced.
- *Send Queue Drain (SQD)*: In this mode send operations will halt except for those that were already underway.
- *Send Queue Error (SQE)*: This state is entered if a completion error occurs when processing an outgoing send request. For Reliable Connection (RC) this state is not used and the QP will directly enter the ”Error State” instead.
- *Error*: In this state the QP is “broken” and all requests that have errored out and not completed will be placed onto the CQ.

### 11.1.2 Reliability Semantics

The reliable transports of InfiniBand: Reliable Connection (RC), eXtended Reliable Connection (XRC) and Reliable Datagram (RD) provide certain reliability guarantees. As

with other transports, there is both a variable CRC and end-to-end CRC that makes sure that data is not corrupted in transit across the network.

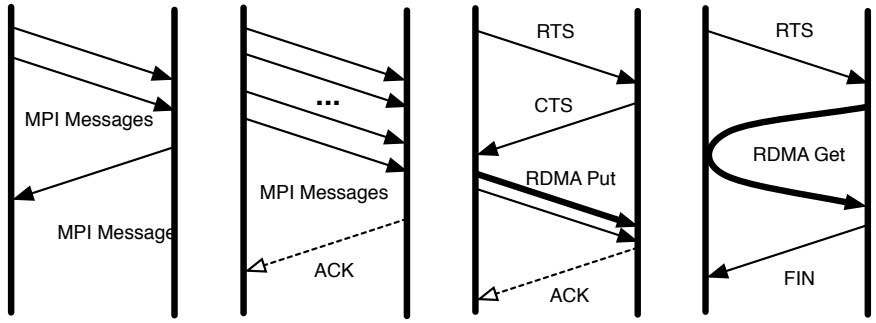
Further, the reliable transports of InfiniBand guarantee ordering and arrival of messages sent. When a message is placed into the Send Queue (SQ), it will not be placed into the Completion Queue (CQ) until the complete data has arrived at the remote HCA. Therefore, upon send completion the sender knows that the data has reached the remote node, but not that the data has arrived in the memory of the other node. This is an important aspect to note. It is not sufficient simply to wait for a send completion if the sender wants to make sure that data has been placed into the end location. Errors on the HCA or over the PCIe bus may prevent data completion.

Further, the lack of a send completion on the sender does not imply that the receiver has not received the data. For example, if before the receiver can send the ACK of a message there becomes a network partition the sender QP will move to an error.

Note that InfiniBand does provide an Automatic Path Migration (APM) feature that can automatically failover to another network path. Using this approach, however, would not be able to try more than one path since after one more failure the QP will again fall into an error state.

### 11.1.3 Error Notification

Errors in InfiniBand are provided both via asynchronous events as well as through the CQ. In many cases the errors will eventually be provided to the CQ, where they can be pulled out and reprocessed if needed. When an error occurs the QP moves into the Error state and must be moved to the Init stage to become usable again.




(a) Eager Piggyback (b) Eager ACK (c) RPUT (d) RGET

Figure 11.2: Reliability Protocols

## 11.2 Design

The goal of our design is to provide a *resilient* MPI for InfiniBand that can survive both general network errors as well as failure of even the HCA. Further, our goal is to keep the overhead of providing this resiliency at a minimum.

As noted in Section 11.1, the InfiniBand network provides certain guarantees and error notifications when using reliable transports. These guarantees, however, are not enough for many MPI applications.

### 11.2.1 Message Completion Acknowledgment

Since InfiniBand completions are not a strong guarantee of resiliency, we must provide message acknowledgments from within the MPI library. Unlike our previous work that showed reliability for unreliable transports in Chapter 5, there is no need to record timestamps and retransmit after a timeout though. InfiniBand will signal an error to the

upper-level if there has been a failure. With this level of hardware support, the acknowledgment design can have lower overhead. Thus, we only need to maintain the messages in memory until a software-level acknowledgment is received from the receiver.

MPI is generally implemented with two protocols: Eager and Rendezvous. Eager is generally used for smaller messages and can be sent to the receiver without checking if the corresponding receive has already been posted. Rendezvous is used when the communication must be synchronized or there are large messages. In the following paragraphs we describe our designs for these two protocols:

**Eager Protocol**

For the eager protocol we follow a traditional mechanism to signal message completion that is common in flow control protocols [29]. The acknowledgment can be piggybacked on the next message back to that process. As shown in Figure 11.2(a), this requires no additional messages. If the communication is not bi-directional, then after a preconfigured number of messages or data size an explicit acknowledgment is sent as shown in Figure 11.2(b). Thus, very little additional messages are required.

Since there are no timeouts, the only additional cost in providing this support is to hold onto the message buffers longer. In the case of a normal Eager design over InfiniBand, a copy of the of data into registered memory is already made, so the send operation has already been marked complete.

**Rendezvous**

When transferring messages, the usual process of data transfer is to use a zero-copy protocol. These are generally classified into two categories: RPut and RGet. We examine reliability designs for each of these methods.

- *RPut*: In this mode the sender notifies the receiver with a Request to Send (RTS) message and the receiver will respond with a Clear to Send (CTS) message that includes the address where the message should be placed. The sender can then perform an RDMA write operation to directly put the message into the memory of the receiver. Normally the sender can mark the send complete as soon as the RDMA write completion is placed in the CQ. With our strengthened semantics though, it cannot be freed until the receiver sends an explicit acknowledgment. (Figure 11.2(c))

- *RGet*: With this protocol the sender sends the source address in the RTS message. The receiver can then do an RDMA read to directly place the data from the source into the destination buffer. Then the receiver then sends a finish message (FIN) to the sender. There is no need for an additional ACK in this case. Thus, there should be no additional overhead. (Figure 11.2(d))

### 11.2.2 Error Recovery

There are two main forms of errors that can occur to the MPI library. These can be classified as Network Failures or Fatal Events. Figure 11.3 shows the basic recovery flow that is attempted in the design.

**Network Failure**

In this case the error is internal to the network. It may come in the form of a "Retry Exceeded" error, which can denote either a cable or switch failure or even severe congestion. In this case we should be able retry the same network path again since it may be a transient issue.

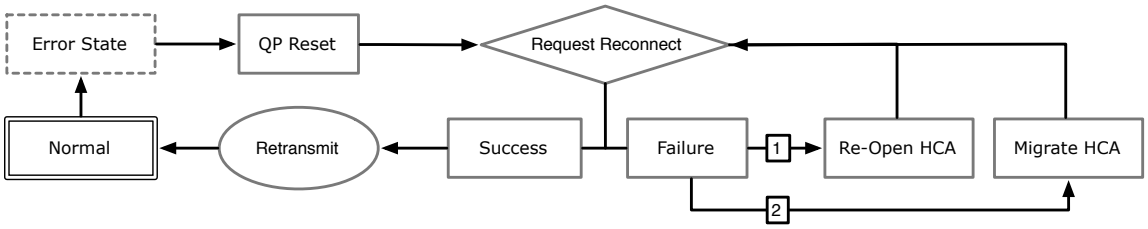



Figure 11.3: Recovery Flow Chart: Upon failure, a reconnect request will take place out-of-band. If this fails or if a fatal HCA event was received then attempt to reopen the HCA or switch to another HCA.

To attempt to reconnect, each side will attempt to send another out-of-band message to the other side. In our prototype, this is done using the Unreliable Datagram (UD) transport where QPs do not fall into the error state due to network errors. If the remote peer replies then the QPs can be repaired and the messages that have not arrived can be re-transmitted. If the remote peer does not respond it could be an internal network failure so we can attempt different paths within the network, with timeouts for each. After a pre-configured length of time the job could be configured to fail.

**Fatal Event**

If we receive an asynchronous event from the InfiniBand library of a "Fatal Event" from the HCA meaning it is not able to continue, we attempt to reload the HCA driver. In this case we must unload all InfiniBand resources (QPs, unpin memory and others). The resources must be freed since after a driver reload these resources are not valid. After reloading we must recreate most of these resources. Some resources, such as QPs, will be re-created once communication restarts by the connection manager. If the same HCA is

not available after a driver reset the library should move connections over to another HCA, if available.

We will then attempt to re-connect with the remote peer. The remote peer will have to reload the connections and memory, but this can be conveyed with control messages.

Note that the sender will interpret a fatal event on the receiver as a Network Failure, so the receiver must reconnect to the sender. In the case of fatal events on both the sender and receiver, reconnect requests must be sent through the administrative network since no InfiniBand resources will be available for either side to reconnect.

## 11.3 Experimental Setup

Our experimental test bed is a 576-core InfiniBand Linux cluster. Each of the 72 compute nodes have dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of 8 cores per node. Each node has a Mellanox MT25208 dual-port Memfree HCA. InfiniBand software support is provided through the OpenFabrics/Gen2 stack [55], OFED 1.3 release.

We have implemented a prototype our design over the verbs interface of the OpenFabrics stack [55]. Our prototype design is designed within the MVAPICH MPI library [53]. Our design uses only InfiniBand primitives for re-connection and can recover from network failures and fatal events. Currently the prototpye does not recover from simultanuous fatal events on communicating nodes, which should be extremely rare, but the design described can withstand those events if fitted with an additional TCP/IP out-of-band channel.

We have verified the correctness of our prototype by using both specialized firmware to introduce errors, defective switches and directly moving QPs to error states.

## 11.4 Experimental Evaluation

In this section we describe the evaluation of our prototype design. We first present an evaluation on both microbenchmarks and application kernels. Then we examine the the cost of recovering from an error using this design.

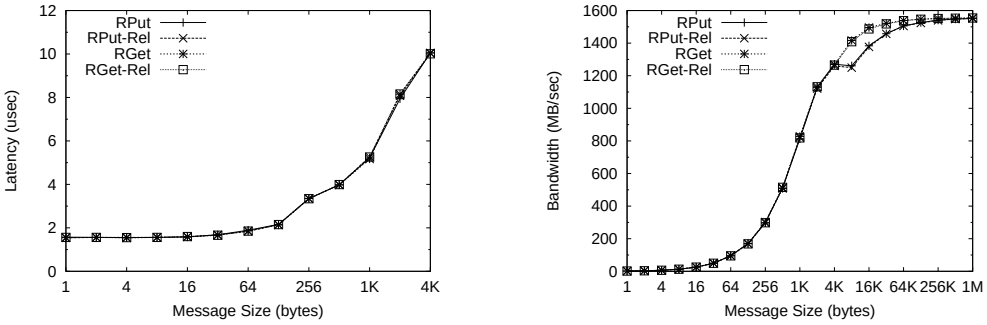
### Methodology

In this section we evaluate each of the following combinations to determine the cost of reliability:

- **RPut-Existing** (RPut): This configuration is the base MVAPICH 1.1 version with the RPut rendezvous protocol.
- **RGet-Existing** (RGet): Same as RPut-Existing, but using the RGet protocol.
- **RPut-Resilient** (RPut-Rel): In this configuration we use the modified MVAPICH version that uses the design from Section 11.2 with the RPut protocol.
- **RGet-Resilient** (RGet-Rel): This configuration is the same as Resilient-RPut, but uses the RGet protocol instead.

### Microbenchmark Performance

We performed an evaluation of base-level microbenchmarks to determine if our resiliency design incurs any overhead. We found that for all standard microbenchmarks including latency, bandwidth and bi-directional bandwidth there is no overhead. Results for latency and bandwidth are shown in Figure 11.4.




(a) Latency (b) Bandwidth

Figure 11.4: Microbenchmark Results

### Application Benchmarks

In this section we evaluate the performance of all of our configurations on the NAS Parallel Benchmark suite to expose any additional overhead.

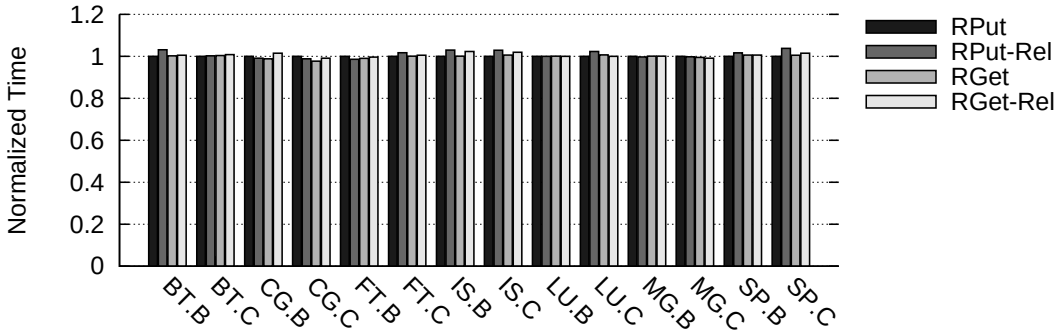



Figure 11.5: NAS Parallel Benchmarks Performance (256 processes)

Figure 11.5 shows the results of our evaluation of each of the NAS benchmarks. All of these results are with 256 processes on classes B and C, as indicated on the graph. The results show very low overhead, with nearly unnoticeable overhead in the RGet-Rel cases.

Table 11.1: Comparison of average maximum memory usage per process for message buffering

| Class | BT.C | CG.C | FT.C | IS.C | LU.C | MG.C | SP.C |
|---|---|---|---|---|---|---|---|
| Reliable | 3.46 MB | 6.92 MB | 0.07 MB | 0.01 MB | 1.65 MB | 0.16 MB | 5.33 MB |
| Normal | 3.20 MB | 6.48 MB | 0.07 MB | 0.01 MB | 1.53 MB | 0.13 MB | 4.95 MB |
| Difference | 0.26 MB | 0.44 MB | 0.00 MB | 0.00 MB | 0.12 MB | 0.03 MB | 0.38 MB |

The RPut-Rel configuration, however, has higher overhead. Recall from Section 11.2, that a reliable form of RPut requires an additional acknowledgment message. For SP.C, which has many large messages, the overhead reaches 4% with only a 1.5% overhead for RGet-Rel. From these results we can determine that the RGet-Rel configuration should be the default configuration.

A different potential overhead of our design is the requirement to buffer messages until they are acknowledged. Large messages will take the rendezvous path and will not be buffered, so only smaller messages will take this path. Table 11.1 shows the memory per process on average required for buffering in both the normal case as well as our design. From the table we can observe that no process requires 500 KB of memory for the additional reliability semantics.

**Recovery Cost**

In this initial evaluation we determine the overhead to detect failures and reconnect required connections. For this evaluation we add a series of transitions to the *Error* state within a microbenchmark between two nodes and track the amount of time from setting the failure to response and reconnection.

Our evaluations show that a single reconnection due to an error takes on average 0.335 seconds. This is a very short period of time and is much faster than could be achieved using timeout methods.

## 11.5 Related Work

Other researchers have explored providing reliability at the MPI layer as part of the LA-MPI project [23, 6]. LA-MPI is focused on providing network fault-tolerance at the host to catch potential communication errors, including network and I/O bus errors. Their reliability method works on a watchdog timer with a several second timeout, focusing on providing checksum acknowledgment support in the MPI library. Saltzer et al. [65] discusses about the need for end-to-end reliability instead of a layered design.

Other MPI designs have included support for failover in the past. Vishnu, et. al., showed how to provide failover for uDAPL to other networks if there was a network partition [86]. Again, this work did not address the issue of HCA failure and relied only on the network guarantees of the underlying InfiniBand hardware. Open MPI [20] also provides support for network failover using their Data Reliability component, but this component can occur a very large overhead on performance since it does not take into account any reliability provided by the hardware and is similar to the support in LA-MPI.

# CHAPTER 12

# OPEN-SOURCE SOFTWARE DISTRIBUTION

Designs from this dissertation are being used in MVAPICH, a high-performance MPI over InfiniBand. MVAPICH is very widely used, including the largest InfiniBand cluster to-date: a Sun Consellation cluster, "Ranger," at the Texas Advanced Computing Center (TACC) [81]. This machine includes 62,976 cores on 3,936 nodes. MVAPICH provides MPI-1 semantics and MVAPICH2 provides MPI-2 semantics.

These software releases are in use by over 900 organizations worldwide and are also incorporated into a number of different vendor distributions. It is also distributed in the OpenFabrics Enterprise Edition (OFED). Many of our designs have already been released or will be released in the near future:

- **MVAPICH 0.9.9** (*April 2007*): In this release the coalescing design from Chapter 4 and has been included into the release to enhance scalability and small message throughput.

- **MVAPICH2 1.0** (*September 2007*): A more advanced version of the coalescing design design from Chapter 4 and has been included in this and future releases.

- **MVAPICH 1.0** (*Feburary 2008*): The full Unreliable Datagram (UD) design from Chapter 5 has been released with this version. This design has already been deployed on "Ranger" (TACC) and "Atlas" (Lawrence Livermore National Lab).

- **MVAPICH 1.1** (*November 2008*): For this release the full hybrid design from Chapter 7 with support for both UD and RC transports simultaneously was included. Also, the XRC transport support from Chapter 8 was included in this release.

- **MVAPICH 1.2** (*Summer 2009*): For this release the network reliability design from Chapter 11 is planned.

Additionally, some of the designs, including the message coalescing design have already been adopted by other MPIs, such as Open MPI [82] and HP MPI [26]. The UD-based design is already being discussed by other vendors for possible future integration into their products as well. The work performed for this dissertation will enable applications to execute at even larger scales and achieve the maximal performance.

# CHAPTER 13

# CONCLUSION AND FUTURE DIRECTIONS

This chapter includes a summary of the research contributions of this dissertation and concludes with future research directions.

## 13.1 Summary of Research Contributions

This dissertation has explored the various transports available with the InfiniBand interconnect and investigated their characteristics. Based on this investigation MPI designs were created for each of the transports, three of which had never had MPI designed for them prior to this work. Further, additional designs to lower memory consumption and increase communication-computation overlap were designed and developed.

### 13.1.1 Reducing Memory Requirements for Reliable Connection

In Chapter 4 the causes of memory usage when using the connection-oriented transports of InfiniBand were investigated. This included the widely-used Reliable Connection (RC) transport and showed a breakdown of memory usage. A key component of the memory usage was found to be the number of outstanding send operations allowed on a single connection. As a result, a coalescing method to pack messages was introduced and evaluated.

Although the hybrid MPI design showed a significant decrease in memory usage and increase in performance, some of the protocols over the different transports can be further optimized. In Chapter 10 the procotol for sending small messages using RDMA operations was addressed. A novel bottom-fill technique was used to reduce the memory required for these small message channels by an order of magnitude.

### 13.1.2 Designing MPI for Unreliable Datagram

Although the coalescing method can achieve lowered memory usage for the RC transport, the connection-oriented transports have the inherent limitation of needing one QP per peer. This increases memory usage and it is also found to decrease performance. The performance characteristics of the Unreliable Datagram (UD) transport were investigated in Chapter 5. An MPI design over UD was designed and implemented and evaluated on over 4,000 processors. This showed a nearly-flat memory usage with increasing numbers of processes and also increased performance in many cases.

### 13.1.3 Designing Reliability Mechanisms

Although the RC transport is often considered the highest-performing, Chapter 5 found that this was not always the case. It is also often considered that reliability should be within the network hardware. In many modern interconnects, such as InfiniBand and Quadrics, the most commonly used interface is a reliable one where reliability is provided in the network device hardware or firmware. With this hardware support the MPI can be designed without explicit reliability checks or protocols. In Chapter 6 we examined this common assumption by evaluating designs with Reliable Connection (RC), Unreliable Connection (UC) and Unreliable Datagram (UD) all within the same codebase. We showed that in many cases a

software-based reliability protocol can perform equal or better than those with a hardware-based reliability.

Clusters with InfiniBand are continuing to scale, yet the frequency of errors per node is not dramatically reducing. Thus, as the number of nodes scales, the Mean Time Between Failure (MTBF) is reducing, so jobs are aborting more frequently. In Chapter 11 we investigated how to leverage software techniques and hardware features to allow an MPI library to withstand network failures both within the network and on the node HCAs.

### 13.1.4 Investigations into Hybrid Transport Design

In Chapters 4 and 5 it was shown that both the RC and UD transports each have their own sets of benefits. In Chapter 7 we looked at how multiple transports can be used together. First, in this chapter a detailed analysis of the network hardware and scalability aspects of each transport mode was evaluated. Second, a new MPI design was proposed that can dynamically switch between transports and messaging modes. Then we evaluated the design and showed that it can perform at least as well as the better of UD and RC. In many cases it could outperform a mode in which only transport is used.

### 13.1.5 MPI Designs for eXtended Reliable Connection Transport

In reaction to the memory scalabilty problems for the RC transport, a new transport eXtended Reliable Connection (XRC) was developed by an InfiniBand vendor. In Chapter 8 this new transport was explored along with the new MPI design possibilities that arise from the features of this new transport. Support for XRC was added into the hybrid MPI from Chapter 7 and was evaluated.

### 13.1.6 Designing Communication-Computation Overlap with Novel Transport Choices

With the new XRC transport, it is possible to address another issue that is very pertinent for large-scale clusters. If an application can overlap communication with computation, then higher performance can be obtained. Chapter 9 described a novel technique of designing an MPI library that allows full overlap of communication with computation. This leveraged the new XRC transport to allow this availability.

## 13.2 Future Work

InfiniBand is very feature-rich and has achieved a large-degree of acceptance in the field of high-performace computing. In this disseration many different designs relating to the transports were explored and evaluated. There are still several research areas that are left to be explored.

**Additional Programming Model Support**: While this dissertation has focused on MPI as the primary programming model, there are other models that are being developed for next-generation clusters. These languages include X10 [27], Chapel [19], Fortress [74], UPC [83], and others. At the core, each programming model requires messaging between nodes. These new programming models offer new sets of semantics though that will offer a new set of challenges both in application design as well as the network-support layers for these models.

**QoS Features of InfiniBand**: InfiniBand specifies Quality-of-Service (QoS) features that will be available in the next generation of HCAs and switches. These allow features such as setting bandwidth guarantees and latency guarantees. This rich set of features can allow novel techniques for giving different priorities to different MPI message types,

particularly as the upcoming MPI-3 standard will include non-blocking collectives and a revised Remote Memory Access (RMA) semantics.

# BIBLIOGRAPHY


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

[2] TOP 500 Supercomputer Sites. http://www.top500.org.

[3] A. S. Tanenbaum. Computer networks. Prentice-Hall 2nd ed., 1989, 1981.

[4] ASC. ASC Purple Benchmarks. http://www.llnl.gov/asci/purple/benchmarks/.

[5] ASC. ASC Sequoia Benchmarks. https://asc.llnl.gov/sequoia/benchmarks/.

[6] Rob T. Aulwes, David J. Daniel, Nehal N. Desai, Richard L. Graham, L. Dean Risinger, Mitchel W. Sukalski, and Mark A. Taylor. Network Fault Tolerance in LA-MPI. In *Proceedings of EuroPVM/MPI '03*, September 2003.

[7] Olivier Aumage, Luc Bougé, Lionel Eyraud, Guillaume Mercier, Raymond Namyst, Loïc Prylli, Alexandre Denis, and Jean-François Méhaut. High performance computing on heterogeneous clusters with the madeleine ii communication library. *Cluster Computing*, 5(1):43–54, 2002.

[8] 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.

[9] P. Balaji, S. Bhagvat, D. K. Panda, R. Thakur, and W. Gropp. Advanced Flow-control Mechanisms for the Sockets Direct Protocol over InfiniBand. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 73, 2007.

[10] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, and J.N. Seizovicand Su Wen-King. Myrinet: a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb 1995.

[11] Dan Bonachea. GASNet Specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[12] Ron Brightwell and Arthur B. Maccabe. Scalability Limitations of VIA-Based Technologies in Supporting MPI. In *Fourth MPI Developer's and User's Conference*, 2000.

[13] Ron Brightwell, Arthur B. MacCabe, and Rolf Riesen. Design and Implementation of MPI on Portals 3.0. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 331–340. Springer-Verlag, 2002.

[14] Ron Brightwell and Keith D. Underwood. An analysis of the impact of MPI overlap and independent progress. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 298–305, New York, NY, USA, 2004. ACM.

[15] Peter N. Brown, Robert D. Falgout, and Jim E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing*, 21(5):1823–1834, 2000.

[16] F. Carroll, H. Tezuka, A. Hori, and Y. Ishikawa. MPICH-PM: Design and implementation of zero copy MPI for PM. 1998.

[17] L. Chai, A. Hartono, and D. K. Panda. Designing Efficient MPI Intra-node Communication Support for Modern Computer Architectures. In *Proceedings of Int'l IEEE Conference on Cluster Computing*, September 2006.

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

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

[20] 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.

[21] M. Gerla and L. Kleinrock. Flow control: A comparative survey. *Communications, IEEE Transactions on [legacy, pre - 1988]*, 28(4):553–574, Apr 1980.

[22] Patricia Gilfeather and Arthur B. Maccabe. Connection-less TCP. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 9*, 2005.

[23] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalksi. A Network-Failure-Tolerant Message-Passing System for Terascale Clusters. *International Journal of Parallel Programming*, 31(4), August 2003.

[24] 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.

[25] Adolfy Hoisie, Olaf M. Lubeck, Harvey J. Wasserman, Fabrizio Petrini, and Hank Alme. A general predictive performance model for wavefront algorithms on clusters of SMPs. In *International Conference on Parallel Processing*, pages 219–, 2000.

[26] HP. HP-MPI. http://www.docs.hp.com/en/T1919-90015/ch01s02.html.

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

[28] InfiniBand Trade Association. InfiniBand Architecture Specification. http://www.infinibandta.com.

[29] 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 04)*, April 2004.

[30] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. *SIGCOMM Comput. Commun. Rev.*, 30(4):309–319, 2000.

[31] S. P. Kini, J. Liu, J. Wu, P. Wyckoff, and D. K. Panda. Fast and Scalable Barrier using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. In *Euro PVM/MPI*, 2003.

[32] K.R. Koch, R.S. Baker, and R.E. Alcouffe. Solution of the first-order form of the 3-d discrete ordinates equation on a massively parallel processor. *Trans. AMer. Nuc. Soc.*, pages 65–, 1992.

[33] R. Kumar, A. Mamidala, M. Koop, G. Santhanaraman, and D. K. Panda. Lock-free Asynchronous Rendezvous Design for MPI Point-to-point communication. In *EuroPVM/MPI 2008*, September 2008.

[34] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/ index.html, August 2001.

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

[36] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SuperComputing(SC)*, 2003.

[37] J. Liu, A. Mamidala, and D. K. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. In *International Parallel and Distributed Processing Symposium*, 2004.

[38] J. Liu, A. Mamidala, and D. K. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS 04)*, April 2004.

[39] 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.

[40] J. Liu, A. Vishnu, and D. K. Panda. Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In *SuperComputing (SC)*, 2004.

[41] 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.

[42] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.

[43] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High Performance RDMA-based MPI implementation over InfiniBand. *Int. J. Parallel Program.*, 32(3):167–198, 2004.

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

[45] Amith R. Mamidala, Sundeep Narravula, Abhinav Vishnu, Gopal Santhanaraman, and Dhabaleswar K. Panda. On using connection-oriented vs. connection-less transport for performance and scalability of collective and one-sided operations: trade-offs and impact. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 46–54. ACM Press, 2007.

[46] D. Mayhew and V. Krishnan. PCI Express and Advanced Switching: Evolutionary path to building next generation interconnects. *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, pages 21–29, 20-22 Aug. 2003.

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

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

[49] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, P. R. Woodward, L. J. Shieh, and S. W. White. Very high resolution simulation of compressible turbulence on the ibm-sp system. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 70, New York, NY, USA, 1999. ACM Press.

[50] Myricom Inc. MPICH-MX. http://www.myri.com/scs/download-mpichmx.html.

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

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

[53] Network-Based Computing Laboratory. MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu.

[54] Network-based Computing Laboratory. OSU Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks.

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

[56] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 55, New York, NY, USA, 1995. ACM.

[57] 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.

[58] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55. IEEE Computer Society, 2003.

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

[60] S. J. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117:1–19, 1995.

[61] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high. performance networking on Myrinet. In *IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 1998.

[62] QLogic. InfiniPath. http://www.pathscale.com/infinipath.php.

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

[64] S. Reinemo, T. Skeie, T. Sodring, O. Lysne, and O. Trudbakken. An overview of QoS capabilities in InfiniBand, Advanced Switching Interconnect, and ethernet. *Communications Magazine, IEEE*, 44(7):32–38, July 2006.

[65] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[66] Sandia National Laboratories. Sandia MPI Micro-Benchmark Suite. http://www.cs.sandia.gov/smb/.

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

[68] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J. P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, and Stanley C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 1–12, New York, NY, USA, 2007. ACM.

[69] G. Shipman, T. Woodall, R. Graham, and A. Maccabe. Infiniband Scalability in Open MPI. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[70] Galen M. Shipman, Ron Brightwell, Brian Barrett, Jeffrey M. Squyres, and Gil Bloch. Investigations on infiniband: Efficient network buffer utilization at scale. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[71] Galen M. Shipman, Stephen Poole, Pavel Shamis, and Ishai Rabinovitz. X-SRQ - Improving Scalability and Performance of Multi-core InfiniBand Clusters. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 33–42, 2008.

[72] Galen Mark Shipman, Tim S. Woodall, George Bosilca, Rich L. Graham, and Arthur B. Maccabe. High performance RDMA protocols in HPC. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[73] Rajeev Sivaram, Rama K. Govindaraju, Peter Hochschild, Robert Blackmore, and Piyush Chaudhary. Breaking the connection: Rdma deconstructed. In *HOTI '05: Proceedings of the 13th Symposium on High Performance Interconnects*, pages 36–42, 2005.

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

[75] 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.

[76] 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.

[77] S. Sur, M. Koop, L. Chai, and D. K. Panda. Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand Architecture with Multi-Core Platforms. In *15th IEEE Int'l Symposium on Hot Interconnects (HotI15)*, Palo Alto, CA, August 2007.

[78] S. Sur, M. J. Koop, and D. K. Panda. High-Performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-Depth Performance Analysis. In *Super Computing*, 2006.

[79] S. Sur, A. Vishnu, H. W. Jin, W. Huang, and D. K. Panda. Can Memory-Less Network Adapters Benefit Next-Generation InfiniBand Systems? In *Hot Interconnect (HOTI 05)*, 2005.

[80] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 32–39, New York, NY, USA, 2006. ACM.

[81] Texas Advanced Computing Center. HPC Systems. http://www.tacc.utexas.edu/resources/hpcsystems/.

[82] The Open MPI Team. Open MPI. http://www.open-mpi.org/.

[83] UPC Consortium. UPC Language Specifications, v1.2, Lawrence Berkeley National Lab Tech Report LBNL-59208. Technical report, Berkeley, CA, USA, 2005.

[84] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *IPDPS '02: Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, page 27.2, Washington, DC, USA, 2002. IEEE Computer Society.

[85] Jeffrey S. Vetter and Andy Yoo. An empirical performance evaluation of scalable scientific applications. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[86] A. Vishnu, P. Gupta, A. Mamidala, and D. K. Panda. A Software Based Approach for Providing Network Fault Tolerance in Clusters Using the uDAPL Interface: MPI Level Design and Performance Evaluation. In *Proceedings of SuperComputing*, November 2006.

[87] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 40–53. ACM Press, 1995.

[88] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, New York, NY, USA, 1992. ACM.

[89] Jiesheng Wu, Jiuxing Liu, Pete Wyckoff, and Dhabaleswar Panda. Impact of on-demand connection management in mpi over via. In *CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing*, page 152, Washington, DC, USA, 2002. IEEE Computer Society.

[90] W. Yu, Q. Gao, and D. K. Panda. Adaptive Connection Management for Scalable MPI over InfiniBand. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.