Matthew Small defended this dissertation on November, 5 2012.

The members of the supervisory committee were:

Xin Yuan
Professor Directing Thesis

Valerie Shute
University Representative

Robert Van Engelen
Committee Member

Ashok Srinivasan
Committee Member

The Graduate School has verified and approved the above-named committee members, and certifies that the dissertation has been approved in accordance with the university requirements.
ACKNOWLEDGMENTS

It is only through the support of many friends, family members and colleagues that this dissertation and the culmination of my doctoral studies are possible. I foremost thank my PhD adviser, Dr. Xin Yuan, who has bestowed upon me untold guidance and support throughout our relationship. His expertise were critical in the development and elaboration of the research in this dissertation, and during my arduous and rewarding doctoral studies, his mentoring provided me with insight, focus, and encouragement. Xin patiently provided a sounding board for my ideas, making them stronger by forcing me to overcome his well-formulated opposition and taught me, largely through his example, the art of performing rigorous scientific research. I truly could not ask for a better adviser.

I also thank the other members of my doctoral committee for providing me with excellent feedback to help make my dissertation stronger. In particular, Dr. Valerie Shute provided insightful advice and detailed formatting notes for this document that went well above and beyond her duties as a committee member. Additionally, I would like to thank Dr. Shute for helping to broaden my research interests by allowing me to participate on an education-related research project in the last year of my doctoral studies that proved to be very fruitful and rewarding work.

Most of all, I thank my parents. My mother has been an unwavering source of love and encouragement in my life. Though I am proud to say that she is the loudest cheerleader of my triumphs, I am most grateful for how she has supported me through the tougher periods in my life. When I have been at my lowest, most tired and nearly broken, she has been there to lift me up, soldier me on and help keep me together. For this, and infinitely more, I am eternally grateful. My father has dutifully worked his
whole life to provide for me and my family, and has exemplified honesty, work ethic,
and humility. He invested in my higher education by sponsoring my first two years
in community college after an unexceptional scholastic performance in high school
followed by a year in the workforce. It was largely this investment, and its greater
implications on his faith in me, that inspired me to work hard and achieve during my
undergraduate studies and beyond. He has provided me invaluable counsel on nearly
all the big decisions in my life and helped shape me into the man I am today. Thus,
this dissertation is dedicated to my mother and father.
# TABLE OF CONTENTS

List of Tables ........................................ vii
List of Figures ......................................... viii
Abstract ................................................ x

1 Introduction ........................................ 1

2 Background ............................................. 5
   2.1 Overview of Parallel Architectures ............... 5
      2.1.1 Memory Architecture ........................... 5
      2.1.2 SMP-CMP Cluster Architecture ............... 7
      2.1.3 Network Architecture .......................... 7
   2.2 Overview of MPI .................................. 9
      2.2.1 Communicators, Ranks and Tags ............... 10
      2.2.2 Point-to-point Communication ............... 10
      2.2.3 Collective Communication .................... 11
      2.2.4 One-sided Communication .................... 14
   2.3 Eager and Rendezvous Protocols ................. 15

3 Related Work ......................................... 19

4 Designing efficient RDMA-based small message channels 22
   4.1 MVAPICH RDMA-based small message channels .... 23
   4.2 Proposed techniques to improve channel efficiency ... 26
      4.2.1 Eliminating persistent buffer association .... 27
      4.2.2 Node-shared small message channels ......... 30
   4.3 Performance study ................................ 35
      4.3.1 Memory requirement ........................... 35
      4.3.2 Communication performance .................. 41
   4.4 Chapter Summary ................................ 45

5 Near-optimal Rendezvous Protocols .................. 47
   5.1 Protocol invocation scenarios .................... 48
   5.2 Existing rendezvous protocols and their limitations ... 50
   5.3 Near-optimal communication protocols for large messages ... 53
# LIST OF TABLES

2.1 Select MPI point-to-point communication routines ........................................ 11
2.2 Select MPI collective communication routines .................................................. 12
2.3 Select MPI one-sided communication routines ................................................... 14
4.1 Memory requirement of MVAPICH and the proposed shared message channel ........ 39
4.2 Communication performance ............................................................................. 44
4.3 Overhead of the mutex lock ................................................................................. 44
5.1 Performance on 16 processes (one process per node) ......................................... 65
5.2 Performance on 121/128 processes (8 processes per node) .................................. 65
6.1 Protocols and the situations when they are used ............................................... 71
6.2 Advantages of the integrated protocols over the traditional rendezvous protocol .... 74
6.3 Performance of application benchmark (all times are in seconds) ....................... 85
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The scope of this dissertation</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Memory architectures</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Overview of an SMP-CMP cluster</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>MPI collectives</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>The eager protocol</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>RDMA rendezvous protocols</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>MVAPICH small message channel components</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Modified small message channel components</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Contrast the receiver side buffer ring</td>
<td>29</td>
</tr>
<tr>
<td>4.4</td>
<td>Two SMP nodes connected by a pair of uni-directional shared small message channels</td>
<td>31</td>
</tr>
<tr>
<td>4.5</td>
<td>Detailed view of a shared message channel</td>
<td>32</td>
</tr>
<tr>
<td>4.6</td>
<td>The next buffer pointer in the header</td>
<td>34</td>
</tr>
<tr>
<td>5.1</td>
<td>The ordering of events in two nodes</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Examples of sender-initiated RDMA write-based protocol with sub-optimal performance</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>Examples of sender-initiated RDMA read-based protocol with sub-optimal performance</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>Example of receiver-initiated protocol with sub-optimal performance</td>
<td>54</td>
</tr>
<tr>
<td>5.5</td>
<td>$SS \leq SW &lt; SW + copy(msg) &lt; RS \leq RW$ scenario and the copy_get protocol</td>
<td>56</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.6</td>
<td>$SS \leq SW &lt; RS(\leq SW + copy(msg)) \leq RW$ scenario and the copy_check_put protocol</td>
<td>57</td>
</tr>
<tr>
<td>5.7</td>
<td>Delayed rendezvous protocols for $SS = RS$ scenarios</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>Progress micro-benchmark</td>
<td>60</td>
</tr>
<tr>
<td>5.9</td>
<td>Progress benchmark results</td>
<td>62</td>
</tr>
<tr>
<td>5.10</td>
<td>Progress results for configuration $(10, 20, 20, 1, 40, 9)$</td>
<td>63</td>
</tr>
<tr>
<td>6.1</td>
<td>The timing when the sender and receiver arrive at a communication</td>
<td>70</td>
</tr>
<tr>
<td>6.2</td>
<td>The hybrid protocol</td>
<td>73</td>
</tr>
<tr>
<td>6.3</td>
<td>The receiver-initiated rendezvous protocol</td>
<td>73</td>
</tr>
<tr>
<td>6.4</td>
<td>High level description of the integrated scheme</td>
<td>76</td>
</tr>
<tr>
<td>6.5</td>
<td>Pingpong performance</td>
<td>81</td>
</tr>
<tr>
<td>6.6</td>
<td>Progress benchmark results (Advantage of HYBRID)</td>
<td>82</td>
</tr>
<tr>
<td>6.7</td>
<td>Progress benchmark results (advantages of RECV_RNDV)</td>
<td>83</td>
</tr>
</tbody>
</table>
ABSTRACT

Clusters of Symmetric Multiprocessing (SMP) nodes with multi-core Chip Multiprocessors (CMP), also known as SMP-CMP clusters, are ubiquitous today. Message Passing Interface (MPI) is the de facto standard for developing message passing applications for such clusters. Most modern SMP-CMP clusters support Remote Direct Memory Access (RDMA), which allows for flexible and efficient communication schemes but introduces a new model that can be challenging to exploit. This dissertation research explores leveraging the flexibility provided by RDMA to optimize MPI point-to-point communications for both small and medium to large messages on SMP-CMP clusters. For small messages, a scheme is devised that improves the buffer memory management in the existing RDMA-based small message channel design; and a novel shared small message channel design is developed that reduces both individual channel resource requirements as well as the number of channels needed by an MPI application, greatly improving small message channel resource utilization and scalability without adding significant overheads or sacrificing the performance benefits of RDMA. MPI medium and large messages are realized by various rendezvous protocols whose performance is very sensitive to the timing of the critical events in the communication (protocol invocation scenarios). As such, existing MPI implementations that use a fixed protocol across all communications suffer from various performance problems such as unnecessary synchronization and communication progress issues. In my research, I explore the idea of protocol customization that allows different protocols to be used for different situations. First, a repository of protocols that can collectively provide near-optimal performance for all protocol invocation scenarios is developed. This repository provides the foundation for profile-driven and compiler-assisted pro-
protocol customization for performance improvement. Furthermore, a communication system with dynamic protocol selection is developed that integrates four protocols into a single system and is able to choose an optimized protocol to suit the run-time characteristics of a particular communication while fully supporting MPI semantics. These techniques reduce unnecessary synchronizations, decrease the number of control messages that are in the critical path of communications, and improve the communication progress, which results in a significantly better communication-computation overlap capability.
Clusters of Symmetric Multiprocessing (SMP) nodes with multi-core Chip Multiprocessors (CMP), also known as SMP-CMP clusters, are ubiquitous today. The majority of the supercomputers in the Top 500 supercomputers list [50] are SMP-CMP clusters. Due to their high performance-to-cost ratios, SMP-CMP clusters are likely to be the dominating high performance computing platforms for the foreseeable future. Synchronization between processes and inefficient data movement across a relatively slow network interconnect can severely impact parallel application performance on SMP-CMP clusters. For this reason, communication optimization is of paramount importance in such environments.

Message Passing Interface (MPI) is the de facto communication standard for high performance computing applications and MPI libraries have been implemented for nearly all modern parallel computing platforms including SMP-CMP clusters. MPI provides precise communication semantics while allowing room for highly optimized implementations. Optimizing MPI point-to-point communication is a priority since most communications in MPI applications are performed by such operations.

Remote direct memory access (RDMA) provides ultra-low latency data-movement directly from the address space of one process to the address space of another without the need for intermediate copies via a network interface card (NIC) that supports RDMA. Most popular high performance computing (HPC) NICs have support for RDMA including Infiniband [16], Myrinet[5], and Quadrics [46]. Though RDMA can provide many performance advantages, its one-sided communication model is differ-
ent from traditional two-sided send/receive communications; optimally exploiting it within the MPI library design is still an open problem and a primary focus of this dissertation research.

MPI users are generally concerned about message latency for small messages and the effective use of memory and network bandwidth for large messages [15, 24]. As a result, MPI point-to-point communication is realized with two classes of internal protocols: eager protocols are used for small messages and make use of internal MPI buffering to push messages quickly from sender to receiver; and rendezvous protocols, used for medium to large messages, negotiate before transferring message data in an autonomous zero-copy fashion that is designed to promote overlap of communication and computation. Due primarily to the one-sided nature of RDMA operations, an RDMA-based small message channel, which manages internal buffers associated with the eager protocol, must face various design constraints that are less difficult to overcome or non-existent in the traditional two-sided send/receive based approach. There is only one working RDMA-based small message channel design, which is used in the MVAPICH MPI library and has limitations including large memory requirements and scalability issues [49, 53]. There are also performance issues with rendezvous protocols including unnecessary synchronizations, problems with communication progress, and limited opportunities for overlapping communication and computation, that have been observed in studies [1, 19, 45]. Most of the performance issues in rendezvous protocols are related to the relative timing of MPI routine calls on the sending and receiving side of the communication being highly correlated to protocol performance; different rendezvous protocols work better in different situations. I will call the relative timing of MPI routine calls in a communication the protocol invocation scenario. This dissertation postulates the following theses:

1. For small messages, a more efficient and scalable RDMA-based small message channel can be created using alternative design strategies while preserving the low-latency communication provided by RDMA.

2. For medium to large messages, it is possible to craft customized rendezvous
protocols such that, for any given protocol invocation scenario, near-optimal performance can be achieved.

3. It is possible to use information about the protocol invocation scenario at runtime to dynamically customize communication for performance improvement.

To verify the first thesis, a scheme is developed that avoids RDMA-based small message channel design components that increase resource utilization and restrict overall design flexibility; building upon this scheme, a novel shared small message channel design is put forth that reduces both individual channel resource utilization as well as the number of small message channels required by an MPI application. To verify the second thesis, a repository of protocols is developed that can collectively provide near-optimal performance for all invocation scenarios, which allows near-optimal performance to be achieved when the appropriate protocol is used for each communication. To verify the third thesis, a dynamic protocol selection system is designed that integrates four protocols into a single system and is able to choose an optimized protocol to suit the run-time characteristics of a particular communication while fully supporting MPI semantics. In all three research approaches, I implement the proposed techniques within a prototype MPI library for Infiniband and gather experimental results by testing the designs on SMP-CMP clusters using both micro and application sized benchmarks.

The scope of this dissertation is indicated in Figure 1.1; I aim to develop techniques to optimize the performance of point-to-point MPI communications on SMP-CMP clusters with RDMA-enabled interconnects. The contributions from the research associated with the first thesis can be classified in the area of MPI small message channel design and optimization [24, 49, 53, 48]. My research approach to the second and third theses contribute to the areas of rendezvous protocol design [54, 45, 10, 47] and more generally, MPI large message communication optimization [21, 20, 25, 57, 1, 54, 22].

This dissertation is organized as follows. Chapter 2 provides an overview of parallel architectures, MPI, and the eager and rendezvous protocols. Chapter 3 describes
related work in the area of MPI point-to-point communication optimizations. Chapter 4 presents my techniques to optimize the design of RDMA-based small message channels. The repository of protocols that can provide near-optimal performance for all protocol invocation scenarios is described in Chapter 5. The dynamic protocol selection approach for medium and large messages is elaborated in Chapter 6. Finally, Chapter 7 provides a summary of the contributions made by this research.
CHAPTER 2
BACKGROUND

2.1 Overview of Parallel Architectures

2.1.1 Memory Architecture

A memory architecture specifies the abstraction of memory presented by the hardware to a parallel program. This includes both the memory layout and speed at which memory can be accessed from a given processor in the parallel system. There are three types of memory architectures generally used for parallel computing: shared memory, distributed memory and hybrid distributed-shared memory.

A shared memory architecture is shown in Figure 2.1a. In this type of memory architecture, all processors share the same global view of main memory; a write by one processor to a location in memory will subsequently be visible to another processor that performs a read from the same location. Thus, a shared memory architecture provides hardware support for implicit communication via memory reads and writes between processors in the parallel system. In actuality, shared memory architectures often have some parts of memory that do not share a direct physical connection with every processor in the parallel system. Examples of this include processor caches [17, 9] and shared memory systems with physically distributed memory [59]. In such systems, the hardware must perform coherency functions to provide the abstraction of a single globally addressable memory.

If the speed at which any processor in a parallel system can read or write data is constant, independent of the memory location, the memory architecture is said to be
consistent with a uniform memory access (UMA) design. This is the view presented in Figure 2.1a. Alternatively, if the speed at which a memory read or write can be performed depends on where data is located relative to a processor (i.e. local or non-local), the memory architecture uses a non-uniform memory access (NUMA) design. Shared memory architectures may have either UMA or NUMA times.

Figure 2.1b depicts a distributed memory architecture. Processors in this architecture type do not share a global address space and instead, each processor has a local memory that is encapsulated from the rest of the parallel system. Data exchange and synchronization between processors are achieved only through explicit communication and thus there is no need for cache coherency. Distributed memory architectures are inherently NUMA systems since accessing local memory locations is faster than accessing non-local locations.

Hybrid distributed-shared memory architectures are the most ubiquitous of the three memory architectures. Figure 2.1c illustrates the layout of a hybrid distributed-shared memory architecture. These architectures use a two-tiered design in which individual nodes consist of processors using a shared memory architecture and multiple nodes are connected to form a larger distributed system. SMP-CMP workstation clusters are the quintessential example of such a system.
2.1.2 SMP-CMP Cluster Architecture

A chip multiprocessor (CMP) is processor design where multiple independent compute cores are integrated on the same physical die. In the past, chip designers have relied on increasing clock speed and instruction-level parallelism to improve CPU performance. This traditional path to CPU design has largely been closed in the last decade due to issues with power consumption and heat dissipation. The result is that major chip manufacturers have been producing CMPs with an ever greater number of compute cores.

A symmetric multiprocessor (SMP) system utilizes a shared memory architecture where multiple identical processing elements are attached to a single shared main memory. Additionally, a single operating system is used to manage the processing elements. Thus an SMP-CMP node is a system consisting of one or more CMPs where the operating system is able to assign a task to any compute core on the node no matter where the data for that task is located in memory.

Multiple SMP-CMP nodes are often physically connected via a network fabric to form a compute cluster that is able to work cohesively on a parallel task. Figure 2.2 shows a high-level view of an SMP-CMP cluster. As parallel tasks inherently need to exchange data, communication between processes running on SMP-CMP clusters is important. There are two forms of communication associated with SMP-CMP clusters: intra-node and inter-node. Intra-node communications move data between processes executing on the same SMP-CMP node. Inter-node communications move data between processes executing on different SMP-CMP nodes and the data must be marshaled through the network fabric connecting the cluster.

2.1.3 Network Architecture

Contemporary interconnects used in high performance computing (HPC) systems, such as Infiniband and Myrinet, typically support both channel and memory semantics. In channel semantics, communications are performed using the traditional two-sided send/receive operations. To transfer a message, the receiving process posts a
receive operation specifying the destination buffer and who the sender is; and the sending process posts a send operation specifying the source buffer and who the receiver is. The interconnect then handles matching sends with receives and performing data movement without the need for further action by either process. Note that sends and receives are matched in the order they are posted (FIFO).

Memory semantics utilize one-sided remote direct memory access (RDMA) operations to allow a process to directly access the address space of another process. The basic RDMA operations are write and read; a sender can post an RDMA write operation to push data to a receiver-side buffer or the receiver can post an RDMA read operation to pull data from a sender-side buffer. RDMA operations allow for more flexible communication than traditional send/receive primitives since the data transfer can be initiated from either the sender or the receiver without involvement of the other party. Additionally, RDMA can offer lower latency communications because the interconnect is not responsible for message matching.

Contemporary interconnects also support a feature called OS bypass that allows communications operations to be posted to the NIC directly from user-space, without the need for operating system intervention. Operating system involvement in communication adds expensive context switching overheads and thus its avoidance
can significantly improve communication performance. However, since user-space processes generally use virtual addressing with demand paging and the NIC works directly with the memory controller using physical memory addressing, this feature requires the user buffer to be registered before being used in the communication. Buffer registration can introduce significant overheads and is a prime target for optimization.

2.2 Overview of MPI

Message passing interface (MPI) [26] is a message passing library specification that emerged from an effort, spearheaded at the Supercomputing 92 conference, to conglomerate and build upon concepts found in numerous existing communication libraries (e.g. P4, PICL, PVM, LAM) to form a common message passing communication standard. MPI primarily follows the message passing programming model, in which processes have local address spaces and data is moved from the address space of one process to that of another through cooperative operations. MPI provides an application programmer interface (API) with precise communication semantics and language bindings for FORTRAN, C, and C++.

The portability and scalability of MPI have largely contributed to it becoming the de facto communication standard for high performance computing applications and MPI libraries have been implemented for nearly all modern parallel computing platforms including SMP-CMP clusters. On SMP-CMP clusters, an MPI program generally consists of one or more autonomous processes executing their own code which communicate with each other through calls to MPI communication primitives. The code executed by each MPI process is often identical in practice, though it need not be. In this way, MPI is consistent with a multiple instruction-stream multiple data-stream (MIMD) model of execution.
2.2.1 Communicators, Ranks and Tags

MPI introduces the concept of a communicator to provide scope for communication operations. Communicators come in two varieties: intra-communications and inter-communicators. Intra-communicators primarily consist of an ordered group of MPI processes and a communication context. Each process in the group is assigned a rank to uniquely identify it within the group. The communication context serves as a way of partitioning the global communication space. Only communications that share the same context can effect each other. MPI provides the universal communicator MPI_COMM_WORLD that consists of all the processes available for communication at the program start. Inter-communicators are similar to intra-communicators but are less commonly used. Inter-communicators provide communication scope for a process from one ordered group to communicate with a process from another ordered group.

Some MPI communications operations accept a user-specified attribute called a tag. The tag attribute is a communication-specific value assigned by the MPI application programmer and can be used to provide finer grain control over matching within a communicator context. Since only communications that use the same tag can effect each other, tags further partition the communication space within a context. Thus the tag can be thought of as providing a sub-context for a communication operation.

2.2.2 Point-to-point Communication

The basic MPI point-to-point operations are send and receive, which copy message data from a buffer associated with the sender to that of receiver. Table 2.1 shows examples of MPI point-to-point communication routines. There are two modes specified for MPI point-to-point operations: blocking and non-blocking. In a blocking point-to-point operation, the MPI call does not return until the communication is complete. Completion of a communication is defined as the point when it is safe to modify the send buffer (send) or use the data received (receive).
Table 2.1: Select MPI point-to-point communication routines

<table>
<thead>
<tr>
<th>Communication Mode</th>
<th>MPI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Send [40]</td>
<td>MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)</td>
</tr>
<tr>
<td>Blocking Receive [36]</td>
<td>MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)</td>
</tr>
<tr>
<td>Non-Blocking Send [34]</td>
<td>MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)</td>
</tr>
<tr>
<td>Non-Blocking Receive [33]</td>
<td>MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)</td>
</tr>
<tr>
<td>Wait [41]</td>
<td>MPI_Wait(MPI_Request *request, MPI_Status *status)</td>
</tr>
</tbody>
</table>

In non-blocking point-to-point operations, the interface is decomposed into routines for starting the communication (e.g. MPI_Isend, MPI_Irecv) and blocking for communication completion (e.g. MPI_Wait). The purpose for decomposing the communication operation into the aforementioned routines is to allow an MPI process to begin a communication operation, perform useful computation while the operation is executed and avoid blocking until absolutely necessary. This concept is known as communication-computation overlap.

MPI point-to-point communications are matched by a tuple of three attributes: communication context (communicator ID), rank, and tag. Note that communication matching is not dependent on size and the receiver may post a buffer equal to or greater than the size of that posted by the sender. If multiple communications are posted with identical attributes, matching is done in FIFO fashion. MPI matching is deterministic and messages will be matched in the order they are posted, regardless of the order in they are delivered by the network fabric.

2.2.3 Collective Communication

MPI collective operations allow coordinated communication among a group of MPI processes. To execute a collective communication, all participating processes
Table 2.2: Select MPI collective communication routines

<table>
<thead>
<tr>
<th>Communication Mode</th>
<th>MPI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier Synchronization [29]</td>
<td>MPI_BARRIER(MPI_Comm comm)</td>
</tr>
<tr>
<td>Broadcast [30]</td>
<td>MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)</td>
</tr>
<tr>
<td>Gather [31]</td>
<td>MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm)</td>
</tr>
<tr>
<td>Scatter [39]</td>
<td>MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvnt, MPI_Datatype recvtype, int root, MPI_Comm comm)</td>
</tr>
<tr>
<td>All-to-All (Scatter + Gather) [28]</td>
<td>MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)</td>
</tr>
<tr>
<td>Reduce [37]</td>
<td>MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)</td>
</tr>
<tr>
<td>Scan [38]</td>
<td>MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)</td>
</tr>
</tbody>
</table>
in the group must call the communication routine with matching arguments. There are three types of collective communication operations: data movement, computation and synchronization.

The MPI collective data movement operations exchange data among a group of MPI processes using pre-defined patterns. The primary patterns used by these operations are depicted in Figure 2.3a. The broadcast, scatter and gather patterns are similar in that they designate one process, called the root, from which the data are distributed or to which the data are coalesced. In the broadcast pattern, data from the root process are distributed to all other MPI processes participating in the operation. The scatter pattern is similar, except that the data buffer at the root is an array and a different element of the array is distributed to each participating MPI process. The gather pattern is simply the reverse of scatter; data from each participating MPI process are coalesced into an array at the root. The allgather pattern also coalesces data from each process into an array, however there is no root and all processes finish the operation with a copy of the complete array. Lastly, the alltoall pattern behaves as though each process performs a scatter, thus each process ends up with an array containing a unique element from every process in the group.

MPI collective computation operations perform a global reduction (e.g. sum, product, max, etc.) on data distributed across a group of MPI processes. The reduction operation is commonly chosen from a list of pre-defined operations provided by MPI but it can also be user-defined. Figure 2.3b shows the patterns used by MPI collective computation operations, namely reduce and scan. The reduce pattern applies the chosen computation operation on data distributed across the participating MPI processes and stores the result at the root. The scan pattern takes into account the MPI rank of each process and stores the result of a partial, but increasingly more complete, reduction at each process.

MPI provides communicator-level synchronization through the MPI_BARRIER routine. Any process calling this routine is blocked until all other processes in the associated communicator group also call this routine. Table 2.2 shows the function
Table 2.3: Select MPI one-sided communication routines

<table>
<thead>
<tr>
<th>Communication Mode</th>
<th>MPI Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Window [42]</td>
<td>MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)</td>
</tr>
<tr>
<td>Put [35]</td>
<td>MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)</td>
</tr>
<tr>
<td>Get [32]</td>
<td>MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win)</td>
</tr>
<tr>
<td>Accumulate [27]</td>
<td>MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)</td>
</tr>
</tbody>
</table>

prototypes for MPI barrier synchronization and for other MPI collective operations.

2.2.4 One-sided Communication

MPI-2 is an extension to the original MPI specification (MPI-1) and includes support for a new set of Remote Memory Access (RMA) communication routines. These routines allow a contiguous region of the local address space within an MPI process to be designated as a window. After initialization, a window can be the target for one-sided communication operations. In one-sided communication operations, an origin MPI process may autonomously perform data reads or writes within a window located on a target MPI process. Since the nature of these operations is one-sided, only the origin MPI process must call an MPI routine specifying all parameters for both the sending and receiving side of the communication.

There are three one-sided communication operations defined by the MPI-2 standard: put, get and accumulate. A put operation allows the origin process to write data from a local memory buffer directly to a window on the target process. Conversely, a get operation allows the origin process to read data from a window on the
target process into a local memory buffer. In an accumulate operation, the origin process supplies a local buffer with data that is combined with data residing in a window on the target process under a specified operation; the result of this combination replaces the original data on the target process. Table 2.3 shows a selection of MPI communication routines associated with one-sided communication.

The three one-sided operations described above are inherently non-blocking and MPI-2 therefore defines two synchronization modes: active target and passive target. In the active target synchronization mode, both the origin and target MPI processes are involved in the synchronization and must call MPI routines. In the passive target synchronization mode, only the origin process is involved and it calls MPI routines to lock the buffer while it performs one-sided communication calls. Calls to one-sided MPI communication routines must always be associated with one of the two aforementioned synchronization modes.

### 2.3 Eager and Rendezvous Protocols

As discussed in section 2.1.2, communications between processes running on different SMP-CMP nodes are classified as inter-node communications. For such communications, MPI users are generally concerned about message latency for small messages and the effective use of memory and network bandwidth for large messages. As a result, different protocols are designed for small and large messages. MPI inter-node communications are traditionally realized using two internal protocols: the *eager* protocol for small/control messages and the *rendezvous* protocol for large messages.

The eager protocol is characterized by the sender pushing the entire message to the receiver regardless of the receiver-side state. Figure 2.4 illustrates an RDMA-based eager protocol [24]. When the sender arrives at the communication, the entire message is copied into a buffer internal to the MPI library. An RDMA write is then posted using the internal MPI buffer on the sender-side as the source and an internal MPI buffer on the receiver-side as the destination. When the receiver arrives at the
communication, the data are copied from the internal MPI buffer to the user-level receive buffer and the communication is complete.

The simplicity of the eager protocol avoids excessive overheads in the critical communication path, which is crucial for small messages; it also does not require synchronization between the sender and receiver, allowing the sender to progress independently after the copy is complete. Although the eager protocol works well for small messages, it is not suitable for large messages due to high resource requirements, such as receiver-side buffering and the time needed to copy messages between buffers. MPI library implementations have *small message channels* that are responsible for managing resources associated with sending and receiving small messages including the internal library message buffers. For RDMA-based small message channels, these buffers must generally be pinned in physical memory via registration upon channel initialization and cannot easily be reclaimed; developing efficient RDMA-based small message channel management schemes is crucial to provide low-latency communication that is also scalable. The only working RDMA-based small message channel design is found in the MVAPICH MPI library. As noted in several studies [49, 53], MVAPICH RDMA-based small message channels have a number of limitations including its large memory requirement and the scalability issues.

Rendezvous protocols are used for medium and large messages and are characterized by a negotiation between the sender and receiver before the data transfer begins. One of the primary goals of rendezvous protocols is to offload data movement
to the network while allowing the MPI process to perform useful computation (i.e. enable communication-computation overlap). The traditional RDMA write-based rendezvous protocol [24] is shown in Figure 2.5a. In this protocol, the sender initiates the communication by registering the user-level source buffer and sending a SENDER READY packet, the receiver then registers the user-level destination buffer and responds with a RECEIVER READY packet, finally the message data are transferred via an RDMA write and a FIN packet is sent to indicate the completion of the communication. The sender must wait for the communication of the data to complete before it can return from the operation. A sender-initiated RDMA read-based rendezvous protocol proposed by Sur [54] is shown in Figure 2.5b. In this protocol, the receiver responds to the SENDER READY packet with an RDMA read operation. After the RDMA read operation is completed, the receiver sends a FIN packet to the sender and completes the operation. The sender exits the operation after it receives the FIN packet.

In comparison to the RDMA write-based protocol, the read-based protocol eliminates one control message (RECEIVER READY) in the critical path, which may result in better communication progress. However, the RDMA write-based protocol allows the sender to complete the operation after the receiver sends the RECEIVER READY packet (before the receiver completes the operation), which can
be done with one MPI routine call that invokes the communication progress engine, while in the RDMA read-based protocol, the sender completes the operation after the receiver receives the data and sends the FIN message, which usually takes two MPI routine calls, one to issue the RDMA read command and the other one to check the completion of the RDMA read operation and send the FIN packet. Hence, depending how the MPI routines are called, there are situations when either protocol performs better. In other words, the ability of any rendezvous protocol to progress and overlap computation with communication is highly dependent on the relative timing of MPI calls on the sender and receiver; I coin this timing as the protocol invocation scenario.
CHAPTER 3
RELATED WORK

An RDMA-based small message scheme must face various design constraints that are less difficult to overcome or non-existent in the traditional two-sided send/receive based approach including explicit buffer management and memory registration, polling-based message notification, and integration of multiple message channels. Liu et al.[24] present a novel resource-limited RDMA-write based small message scheme, that falls back to the send/receive mechanism upon resource exhaustion, used within the MVAPICH MPI library. The scheme provides a significant performance improvement over traditional send/receive based small message schemes but lacks robust scalability largely due to a requirement that persistently associated buffers be pre-allocated on each side of every uni-directional connection. This scalability issue and large memory requirement are noted in several studies [49, 53]

To improve small message channel scalability, it is proposed [49, 53] to leverage the shared receive queue (SRQ) feature of Infiniband that allows receive buffers to be allocated per process and shared across all connections rather than being allocated per connection. Unfortunately, Infiniband SRQ only allows for channel-based communication and therefore does not benefit from the low latency provided by RDMA. The authors of [48] propose a scheme that modifies the eager protocol to remove the need for sender-side data buffering in MPI communications involving frequently-used buffers, however the buffer registration is speculative in nature and can hurt performance when buffers are registered and not subsequently used frequently enough to recoup the large registration overhead.
The performance issues with rendezvous protocols including unnecessary synchronizations, problems with communication progress, and limited opportunities for overlapping communication and computation, have been observed in many studies [1, 19, 45]. Various techniques have been developed to overcome these problems. The techniques can be broadly classified into three types: using asynchronous communication to improve computation-communication overlap [21, 20, 25, 57], using interrupts to improve communication progress [1, 54, 22], and designing better protocols to avoid unnecessary synchronizations [54, 45, 10, 47].

Asynchronous communication progress allows communications to be performed asynchronously with the main computation thread. An asynchronous messaging scheme is presented in [21] that attempts to bridge the gap between the limited in-order matching semantics of Infiniband and the matching requirements of MPI through a novel method of allocating Infiniband receive queues, however this approach is resource prohibitive for MPI programs that use a number of distinct tags. A scheme is discussed in [20] that utilizes NIC programmability provided by Myrinet and also implements MPI matching semantics at the hardware level. The downside of this scheme is that NIC programmability is a feature specific to Myrinet and the flexibility it provides tends to be offset by network performance degradation. An event-driven MPI implementation is presented in [25] that utilizes a helper thread to provide asynchronous communication progress at the software level. The scheme improves computation and communication overlap but incurs a performance penalty for context switches between the application thread and the helper thread, especially for synchronous communications.

The interrupt driven message detection approach allows each party (sender or receiver) to react to a message whenever the message arrives. An interrupt mechanism is utilized in [54] to allow the receiver to asynchronously detect and respond to the arrival of a control message in the RDMA read-based protocol. The scheme introduces significant implementation overheads such as: locking shared data within the critical communication path, using multiple expensive interrupts to make progress for
a single communication, and generating unnecessary interrupts when the application is already making progress. Modifications to the scheme are made in [22] to address some of the aforementioned overheads, however there is still a non-negligible interrupt overhead.

The third approach tries to improve the performance with better protocols, which can benefit both synchronous and asynchronous communications. It is shown in [54] that a sender-initiated RDMA read-based rendezvous protocol uses fewer control messages between the sender and the receiver than the RDMA write-based rendezvous protocol. Pakin [45] demonstrates that the receiver-initiated rendezvous protocol is simpler and can achieve higher performance in most cases than the sender-initiated protocol. The Gravel library [10] is a repository of communication primitives designed to replace select MPI calls at compile-time with optimized, custom-made protocols to suit a particular communication pattern and decouple synchronization from data movement. Rashti [47] proposes a run-time MPI communication scheme combining the sender-initiated read-based rendezvous protocol [54] and the receiver-initiated protocol [45]. This scheme shows the potential benefit of dynamically selecting protocols at run-time, however it does not support MPI wild-cards and the protocol integration method used introduces excessive control overheads, requiring each control message to be acknowledged.
CHAPTER 4
DESIGNING EFFICIENT
RDMA-BASED SMALL MESSAGE CHANNELS

As detailed in section 2.1.3, inter-node communications on SMP-CMP clusters can be carried out using either the traditional two-sided (send/receive) semantics or the one-sided (RDMA write, RDMA read) semantics. Each type of communication semantics offers advantages in some situations [24, 49]. In general, send/receive primitives are more scalable while RDMA primitives offer lower latency for small messages when the number of channels is not too large.

To achieve better small messaging performance, MVAPICH [24], a widely deployed MPI library for InfiniBand clusters, adopts RDMA-based small message channels when the number of channels needed is small. As noted in several studies [49, 53], MVAPICH RDMA-based small message channels have a number of limitations including its large memory requirement and the scalability issues. MVAPICH addresses these issues by employing a fall-back to send/receive based small message channels when the resources for RDMA-based channels are exhausted [53].

In this chapter, I propose a novel design of RDMA-based small message channels for InfiniBand clusters with SMP nodes that significantly improves the memory requirement and scalability of the design in MVAPICH. First, techniques are developed to eliminate persistent buffer association [24], a component of MVAPICH RDMA-based small message channel design that not only results in significant mem-
ory requirement, but also imposes restrictions in the buffer memory management for the communication channels; by eliminating persistent buffer association, the buffer memory for the channels can be managed much more effectively. Building upon this technique, a novel RDMA-based shared small message channel design is proposed that allows MPI processes on the same SMP node to share RDMA-based small message channels, which significantly reduces the number of small message channels needed for MPI programs on InfiniBand clusters with SMP nodes. In experiments, I quantitatively study the improvement in memory usage and evaluate the impacts of the new design on the communication performance. The results indicate that this novel design greatly reduces the memory required by RDMA-based small message channels and improves the scalability without adding noticeable overheads or sacrificing the performance benefits of RDMA.

The rest of this chapter is organized as follows. Section 4.1 describes the RDMA-based small message channels in MVAPICH. My proposed techniques that improve over the MVAPICH channels are detailed in Section 4.2. The performance study is presented in Section 4.3. Section 4.4 concludes.

### 4.1 MVAPICH RDMA-based small message channels

The details of MVAPICH RDMA-based small message channels can be found in [24]. Here, I will give a brief overview of MVAPICH RDMA-based small message channels with emphasis on the techniques related to my proposed schemes.

MVAPICH RDMA-based small message channels utilize a ring of small message buffers that are allocated and registered on both the sending and receiving sides of every uni-directional point-to-point connection upon library initialization. The buffer rings are illustrated in Figure 4.1a. Registering the buffers outside the critical communication path avoids synchronization between sender and receiver as well as an expensive memory pinning overhead during communication. Using these buffers, a
small message is communicated by (1) the sender copying the message into its buffer, (2) the sender RDMA-writing the message into the buffer in the receiving side, and (3) the receiver copying the message from its buffer to its user space. This is the eager protocol for communicating small messages [24]. Since the sender must know the memory address of the data sink before posting an RDMA write, the order of buffer consumption is controlled on each side by the head and tail pointers. Buffer reclamation is handled by the receiver piggybacking consolidated information about buffers which are safe for reuse on future messages.

To take full advantage of the low latency communication associated with RDMA, an incoming message must be detected by the receiver through memory polling and cannot utilize any hardware notification features provided by InfiniBand. In addition, InfiniBand RDMA messaging start-up overhead makes it only practical to perform a single RDMA operation for each small message communication. To allow detection of incoming messages, a sender packages each small message as follows: the message header including the data size, a head flag that indicates to the receiver the header has arrived, message data, a tail flag indicating completion of the RDMA write. Figure 4.1b shows the small message buffer layout used in this scheme. The receiver polls the head flag to detect an incoming message, then polls the memory location just past the message data (i.e. tail flag) to detect completion of the RDMA write. This polling scheme relies on an undocumented InfiniBand implementation feature that data is written in order from source to sink [24].

Using this polling scheme, the sender must choose a tail flag value that produces a detectable change in the receiving side buffer and the location of the tail flag varies with message size. Unless the receiver clears the entire buffer after every received message, which could significantly degrade the performance, the tail flag value used by the sender may coincide with data already residing in the receiver buffer. To address this issue, the MVAPICH design uses a concept they call persistent buffer association in which each buffer on the sending side of the connection is exclusively paired with a buffer on the receiving side on the connection. Because all buffers are
initialized to zero during MPI_Init and the scheme stipulates that small messages may only be transferred between persistently associated buffers, the buffer pairs contain the same data on both sides of the connection. Thus the sender has local access to the same data contained in the receiving side buffer and is able to select a tail flag that is distinct. Persistently associated buffers are connected by dotted lines in Figure 4.1a.

The primary disadvantage of persistent buffer association is the inflexible use of the buffer memory, which results in ineffective buffer management and utilization. With persistent buffer association, the sender must allocate buffers for a small message channel upon initialization and each buffer is locked, only available for sending messages destined to the associated buffer on the receiver side. Additionally, each buffer must be allocated to handle the largest possible small message, regardless of the actual size of the data (or control message) itself. In other words, resources on the sender side must always be allocated for maximum channel utilization, even when the channel is not busy. The memory requirement with persistent buffer association significantly limits the usage of the RDMA-based small message channels. Moreover,
this requirement of association of each sender buffer with one particular receiver buffer also limits how the memory resources can be managed. For example, both the sender-side and receiver-side buffers can only be freed in the particular buffer ring order.

4.2 Proposed techniques to improve channel efficiency

In this section, I will first present a scheme that eliminates the need for persistent buffer association. This scheme allows the sender side buffers for different messages to (1) be allocated in the exact amount (for the message data and the header) and (2) be shared in a buffer pool, which completely removes the sender side buffer fragmentation associated with persistent buffer association and reduces memory utilization. Furthermore, eliminating the association between the sender side and receiver side buffers enables the buffer memory at the sender and the receiver to be managed independently, facilitating more flexible design of buffer management schemes for small message channels. Building upon this technique, a novel design for shared RDMA-based small message channels where processes in an SMP node share small message channels is designed. Using shared channels, the number of small message channels needed for an MPI program on SMP-CMP cluster is significantly reduced, from $O(n^2)$ where $n$ is the number of MPI processes to $O(N^2)$ where $N$ is the number of SMP nodes for the program. In addition, the shared message channels also improve the receiving-side buffer management. My design greatly improves the scalability of RDMA-based small message channels, allowing such channels to be used by a much larger number of MPI processes in comparison to those in MVAPICH. A performance study shows that the design achieves these improvements without adding noticeable overheads or sacrificing the performance benefits of RDMA.
4.2.1 Eliminating persistent buffer association

As previously discussed, the primary reason that MVAPICH small message channels require persistent buffer association is for the sender to set a flag at the end of the message such that the receiver can poll to determine the completion of the message. Since MVAPICH fixes the starting address of the buffer at the receiver side, the location of the end-of-message flag is undecided (i.e. changes with message size). As a result, a statically chosen tail flag value will be ineffective: theoretically, all data values in the buffer will need to be cleared to avoid the tail flag coincidentally having the same value as the data in the buffer. In MVAPICH, the sender dynamically selects an appropriate tail flag by examining and avoiding the current value at the location of flag at the receiving buffer with persistent buffer association.

My technique is based on the following observation: if the location of the tail flag is fixed regardless of the message size, then the communication system can use a static message flag with the receiver resetting the tail flag after receiving each message. The receiver only needs to reset the tail flag at a fixed location to allow the buffer to be used for future messages. This eliminates the need for persistent buffer association.

My modified design uses a circular ring of buffers allocated and pre-registered on only the receiving side of every uni-directional channel. On the sending-side of the channel, there is a ring of buffer pointers corresponding to buffers on the receiving side. Additionally, each process has a registered-memory pool shared across all channels that can be used to allocate small message buffers for sending data. Head and tail pointers are maintained on both sides of the connection; this allows the sender and receiver to coordinate the order of receive buffer usage and reclamation in a similar fashion to the MVAPICH small message scheme. Figure 4.2a shows the modified data structures. As shown in the figure, the sender has a ring of buffer pointers that point to the registered memory pool that is shared by all messages. For each message, the exact memory amount is allocated for the message instead of the maximum message size.

To transfer a message, the sender allocates a buffer from a registered-memory
pool and packs the small message as follows: the message data, the message header including the data size, and a tail flag. The tail flag may be any non-zero value chosen statically. Figure 4.2b shows the layout of a small message buffer in the modified design on the receiving side (on the sending side, an exact amount of memory is allocated for the message data, the header, and the flag). Once the send buffer is ready, the sender performs an RDMA write of the data from the local library buffer to the destination buffer. Different from the MVAPICH scheme that fixes the starting location of a message (at the beginning of the buffer) at the receiving end, this scheme fixes the ending of a message at the end of the receiving-side buffer so that the message flag is always at the end of a receiving-side buffer: the sender varies the starting message address of the RDMA write to align the tail flag accordingly. Thus the receiver polls the tail flag (initialized to zero) to check for new messages and a change in the tail flag indicates all message data have arrived. The receiver then copies the message data to the user-level receiver buffer, resets the tail flag, and notifies the sender that the buffer is safe for reuse. Figure 4.3 contrasts the receiving-
side buffer ring in MVAPICH and in my proposed scheme. Basically, MVAPICH aligns the message at the beginning of the buffer and has the unused buffer space at the end of each buffer while my system aligns the message at the end of the buffer and has the unused space at the beginning of the buffer. Notice that for small message channels, the message in the user space is always copied to the library buffer before the communication. Hence, the library buffer organization as well as the message format can be manipulated by the library freely without affecting user memory.

As previously noted, each MPI process in the modified design pre-allocates a single pool of registered memory upon library initialization that is used to service dynamic requests for send buffer memory. Memory from this pool can be used to send a message destined to any remote receive buffer and on any message channel. Thus, eliminating persistent buffer association from the small message channel design gives the sender much greater flexibility in allocating small message buffers and reduces memory fragmentation caused by committing these buffers immediately upon channel initialization, as in the MVAPICH design. Additionally, the pool of registered memory can be managed in a malloc-like fashion and used to service exact registered memory allocation requests; this can be most beneficial when sending very small data or control messages.
4.2.2 Node-shared small message channels

Traditionally, RDMA-based small message channels are established between each pair of processes. Thus, for an $n$ processes MPI job, the number of small message channels needed is potentially $O(n^2)$. For MPI applications with a large number of processes, this leads to not only significant memory usage but also increased communication overheads as a receiver needs to individually poll each message channel for potential incoming messages. The number of channels needed can significantly affect RDMA performance for small messages [15]. To address this issue, I propose a shared RDMA-based small message channels design that is built upon the scheme in the previous sub-section. With shared small message channels, multiple processes on an SMP node can share one communication channel for communicating with a set of processes on another SMP node. This reduces the number of channels needed to run an application from $O(n^2)$ where $n$ is the number of processes (cores) to $O(N^2)$ where $N$ is the number of nodes for the application. Additionally, shared small message channels eliminate the logical ring buffer organization at the receiving-side and allow multiple receivers to share the receiving-side buffers, which greatly improves the effectiveness of memory management and utilization, resulting in significant saving in memory requirement for small message channels. Experimental results show that with shared message channels, the memory requirement for small messages for MPI applications is reduced very significantly, sometimes by an order of magnitude.

Figure 4.4 shows two nodes connected by shared message channels. Unlike traditional RDMA-based small message channels where a pair of channels are allocated for each pair of processes in the MPI application, a pair of shared small message channels only need to be allocated for each pair of SMP nodes. This reduces the resource fragmentation and polling overhead that limits the scalability of the traditional RDMA-based small message system design. Note that to manage resource contention, a shared message channel can be established for a subset of the processes in an SMP node to communicate with a subset of the processes in another SMP node. However, to ease elaboration, I describe the technique with the assumption that one
Figure 4.4: Two SMP nodes connected by a pair of uni-directional shared small message channels

shared message channel is used for all processes in one node to all processes in another node.

**Shared message channel organization.** The main idea of the shared message channel is to let the senders to use data structures in shared memory to coordinate and manage the receiving-side buffers, which are pooled for the channel in this design. Figure 4.5 provides a detailed view of a single RDMA-based shared message channel. On the sending-side of the channel, an **available buffer pool** (in sender shared memory) holds references to buffers on the receiving-side of the channel that are ready to receive small messages. In addition to the buffer references held in the available buffer pool, a **next buffer** (in sender shared memory) reference is maintained for each MPI process on the receiving-side of the channel. The buffers pointed to by the next buffer pointer are designated to be used as the destination of the next small message sent to the respective receiving-side process. Because the available buffer pool and next buffer references are shared by all processes on the sending-side of the channel, access to these structures are sequentialized using a single mutex lock. Note that this lock only mediates access to the shared message channel structures (the available buffer pool and the next buffer variables) and need not be held when performing data
Figure 4.5: Detailed view of a shared message channel.
movement. As in the technique discussed in the previous sub-section, each process has a registered memory pool that is used for allocating sending-side buffers for sending small messages on any channel.

On the receiving-side of the channel, a single block of contiguous memory is divided into maximum-sized (i.e. size of the eager threshold) small message buffers used for receiving messages. This memory is allocated and registered by a single MPI process and then made available to the other processes on the node through shared memory mapping. Each process on the receiving-side of the channel maintains a local variable that references the buffer where it expects the next message from the sending-side of the channel and where any sending-side process may deposit a message.

Upon channel initialization, processes on both sides of the shared small message channel allocate resources and exchange addressing information. It is also at this time that initial next buffers at both the senders and the receivers are agreed upon.

**Sending and receiving small messages.** To send a message using a shared small message channel, the sender first acquires the mutex lock protecting the shared data structures at the sending-side of the channel. Once this is done, the next buffer reference corresponding to the receiving process is noted and replaced with a buffer from the available buffer pool. The replacement buffer reference is also noted and the mutex lock is released. Thus, the sender has acquired a remote buffer in which to deposit the small message and noted the buffer that will be used to deposit the next small message sent to the same receiver. Notice that the critical section protected by the mutex lock is very small, there are only a small constant number of instructions needed to modify the shared data structures (the available buffer pool and the next buffer variable).

The sender must now allocate a local small message buffer to store a copy of the small message. It allocates this buffer from its shared registered memory pool (shared by all channels for sending-side buffers). The sender then copies the small message data from the user-level source buffer into the registered library buffer just allocated, prepares the message header, and RDMA-writes to the remote destination buffer; and
the send is complete. The message header contains the message size as well as the
next receiving-side buffer reference so that when the receiver receives the message, it
knows which buffer will be used for the next message to it in the shared channel.

Because buffer management is performed on the sending-side of the shared message
channel, receiving a message from another node is simple. The receiving process
always knows where the next message will arrive from a particular node. Initially,
this is statically set in MPI_Init. When program executes, every time a message is
received, the header of the message has the next buffer field, as shown in Figure 4.6,
to indicate to which buffer the next message to it will be deposited. The receiving
process simply polls each buffer in a set of channels that a message could potentially
arrive from. When a message arrives, the receiving process copies the message data
to the destination buffer and updates the next buffer variable with the field that the
sender placed in the message header. The receive is completed at this point.

**Buffer reclamation.** To allow continuous use of a shared small message channel
there must be a mechanism to replenish the available buffer pool. Since small messages
are received (i.e. copied from the library buffer to the user-level destination buffer) at
a rate independent of the rate at which they are sent, processes on the receiving-side
of the channel must notify the processes on the sending-side when buffers are again
available for use. To this end, each process on the receiving-side of the shared small
message channel maintains an acknowledgment (ACK) list that tracks buffers that
are ready for re-use. Most of the time, similar to MVAPICH, a process will be able to
piggyback the list of consumed messages within the header of another small message
sent in the reverse direction. However, an explicit ACK message can be sent if the
ACK list grows past a predetermined threshold to ensure the channel can continue to function. When an ACK is received, the corresponding buffer references are added back to the sending-side available buffer pool.

Besides the reduction of the number of channels needed for an application, the shared message channels improve channel buffer usage in several ways. First, the sender-side buffers are effectively managed: one pool of buffers are shared among all channels and are managed in a malloc-like fashion for allocating the exact size for each message. In addition, the eliminating of persistent buffer association allows the sending-side buffer to be reused as soon as the RDMA operation completes. Second, the receiving-side buffers are also managed more effectively in comparison to the MVAPICH design: (1) receiving side buffers are shared among multiple processes, and (2) each process only keeps the buffers for active outstanding messages instead of the statically allocated buffer ring regardless of the communication status. In the shared channels, the receivers also have more opportunities to piggyback the ACKs to release the receiver side buffers.

4.3 Performance study

There are two major aspects of the proposed RDMA-based shared message channels that need to be understood: the reduction in memory requirement and the impact on communication performance. My performance study focuses on these two aspects.

4.3.1 Memory requirement

The memory required for the small message channels to run a program is determined by the dynamic program behavior, that is, the maximum number of outstanding small messages during the whole execution of the program. Since practical systems such as MVAPICH allocate a fixed amount of memory for the channels (the communications fall back to the send/receive channels when the resources for the RDMA channels exhaust), it is infeasible to just run the programs and measure the memory requirement. In this study, I trace MPI applications, analyze the traces
to obtain the small channel usage information for each channel, and determine the memory requirement for MVAPICH and the proposed RDMA-based small message channels. The performance metric used for comparison is the average and maximum per-process memory required for running an application. In obtaining this metric, it is assumed that all small messages and control messages are communicated with the RDMA-based small message channels. Note that small messages are messages whose sizes are below the *eager threshold* parameter (EAGER\_THRESHOLD). The typical value for EAGER\_THRESHOLD in a practical system is between 8K bytes and 12K bytes.

The traces are collected on the SDSC Gordon cluster [14], which consists of 1024 compute nodes, each containing two 8-core 2.6 GHz Intel EM64T Xeon E5 processors and 64 GB of DDR3-1333 memory. Groups of 16 nodes are connected to switches by 4x QDR (40 Gbit/s) using a 4x4x4 3D torus network topology with adjacent switches connected by three 4x QDR InfiniBand links. The nodes run CentOS with linux kernel 2.6.34.7-1 and use the mvapich2-1.8a1 MPI library. The eager threshold is 8200 bytes.

Six benchmarks are used in this study. Three benchmarks are from the NAS parallel benchmarks [44]: BT, CG, and SP. The CLASS D problem size is used for all of the three programs. Two programs, Sweep3D [2] and SMG2000 [3], are from the ASCI benchmarks. The program size and related parameters for the ASCI benchmarks are as follows: Sweep3D on 529 processes has a grid size of $920 \times 920 \times 920$ with $mmi = 16$, $mk = 3$. Sweep3D on 1024 processes has a grid size of $928 \times 928 \times 928$ with $mmi = 16$, $mk = 3$. For SMP2000, the problems are specified as $n = 35 \times 35 \times 35$, $P = 8 \times 8 \times 8$, and $c = 0.1$, 1.0, 10.0 for 512 processes runs, and as $n = 35 \times 35 \times 35$, $P = 10 \times 10 \times 10$, and $c = 0.1$, 1.0, 10.0 for 1024 processes runs. Please see the meaning of benchmark parameters in the ASCI benchmark sites [2, 3]. The last program is a matrix multiply (MM) program that multiplies $8192 \times 8192$ matrices 100 times.

TU Dresden’s VampirTrace library (version 5.13.2) [61] is used to gather trace data. The original MPI application code is slightly modified to provide extended
MPI call information via trace comments. The trace analysis tool parses the resulting trace and simulates the messaging behavior of the small message channels. The tool uses standard MPI matching semantics to match point-to-point communication calls and uses timing information gathered from running the ping-pong benchmark on various message sizes to approximate network latency and message transfer times. The trace analysis tool tracks the small message channel memory allocations that are associated with all point-to-point communications for each MPI process, including both the small eager messages and the control messages used in the Rendezvous protocol for large messages. For each small/control message, the tool determines from the trace the timing when buffer memory is allocated (using the time when the communication starts) for the message and when the buffer memory is reclaimed for future uses. For MVAPICH, the buffers remain allocated until a message is sent in the opposite direction, from the original receiver to the original sender, carrying a piggy-backed ACK, as specified in [24]. To have a fair comparison, the proposed shared small message channels also only use piggy-backed ACKs to release receiver side buffers (explicit ACKs are disabled). With the timing information, for each communication scheme, the analyzer computes the maximum memory required for each channel, which is the amount of buffers for the largest possible outstanding messages that are allocated, but not reclaimed, during the whole execution of the programs. In computing the total memory requirement, for both MVAPICH and the proposed scheme, it is assumed that each process has a small message channel to any other process.

All MVAPICH small message channel allocations are for a pair of persistently associated buffers (i.e. two EAGER_THRESHOLD sized buffers) that are associated with a uni-directional process-to-process channel. Since MVAPICH with persistent buffer association pre-allocates the same amount memory for all processes and channels, the average and maximum per-process memory requirement are the same. Let \( M_{C(\text{mea})} \) be the largest memory requirement of any process-to-process channel and \( nprocs \) represent the number of processes for the MPI program. Since each process
must connect to \( nprocs - 1 \) other processes (one of the assumptions), the memory requirement in each process for MVAPICH is thus

\[
M_{MVA} = M_{C(mva)} \times (nprocs - 1)
\]  

(4.1)

In the node-shared small message scheme, sender-side buffer memory is allocated from the process-shared memory pool while receiver-side buffer memory is associated with a uni-directional node-to-node channel. Thus, two allocations are made when sending a small data or control message: a process-shared sender-side allocation of \textit{message\_size} plus the header size bytes and a channel-associated receiver-side allocation of EAGER\_THRESHOLD bytes (maximum size for small messages). The sender-side buffer is free when the communication is completed. The receiver-side buffer is not freed until a message is sent in the opposite direction, from the original receiving node to the original sending node, carrying a piggy-backed ACK. The trace analyzer computes the largest memory requirement of any process-shared sender-size memory buffers and the largest memory requirement of shared receiving side buffers in any channel. Let \( M_P \) be the largest memory requirement for any process-shared sender-size buffers, \( M_{C(shared)} \) be the largest memory requirement of receiving-side buffer of any shared message channel. Let \( nnodes \) be the number of SMP nodes for the MPI job, since each process has a channel to any other process, the maximum per-process memory requirement with my scheme is
Table 4.1: Memory requirement of MVAPICH and the proposed shared message channel

<table>
<thead>
<tr>
<th>Program</th>
<th>NPROC</th>
<th>MVAPICH</th>
<th>Shared channels</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$M_{C(MVA)}$ (KB)</td>
<td>Per-process (MB)</td>
</tr>
<tr>
<td>MM</td>
<td>512</td>
<td>32.8</td>
<td>16.8</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>32.8</td>
<td>33.6</td>
</tr>
<tr>
<td>BT</td>
<td>529</td>
<td>65.6</td>
<td>34.6</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>557.6</td>
<td>570.4</td>
</tr>
<tr>
<td>SP</td>
<td>529</td>
<td>65.6</td>
<td>34.6</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>180.4</td>
<td>184.6</td>
</tr>
<tr>
<td>CG</td>
<td>512</td>
<td>49.2</td>
<td>25.1</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>49.2</td>
<td>50.3</td>
</tr>
<tr>
<td>Sweep3D</td>
<td>512</td>
<td>32.8</td>
<td>17.32</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>885.6</td>
<td>906.0</td>
</tr>
<tr>
<td>SMG2000</td>
<td>512</td>
<td>138760</td>
<td>70969</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>719041</td>
<td>718332</td>
</tr>
</tbody>
</table>
\[ M_{\text{Max:Shared}} = M_P + M_{C(\text{shared})} \ast (\text{nnodes} - 1) \] (4.2)

Let \( nwidth \) be the number of processes in a node. Since \( M_{C(\text{shared})} \) is shared by \( nwidth \) processes, the average per-process memory is

\[ M_{\text{Ave:Shared}} = M_P + \frac{M_{C(\text{shared})} \ast (\text{nnodes} - 1)}{nwidth} \] (4.3)

Note that \( M_{\text{Max:Shared}} \) denotes the maximum amount of memory in the address space of each process that must be allocated for the small message channel while \( M_{\text{Ave:Shared}} \) times the number of processes is the total size of the actual physical memory that is used by all processes for the communication. Since my scheme uses shared receiver side buffers, \( M_{\text{Ave:Shared}} \) is always less than \( M_{\text{Max:Shared}} \).

Table 4.1 compares the memory requirement for MVAPICH and the proposed shared message channels. The results for MVAPICH confirm early studies that show that MVAPICH RDMA-based small message channel requires very large amount of memory for medium sized jobs such as those in the experiments with 500 to 1024 processes. There are some interesting observations for the proposed shared small message channels. First, the sender side process-wide sender buffer pool is very small for all cases: the largest one is only 7200 bytes (SMG2000 with 1024 processes). This is due to the elimination of persistent buffer association that allows the sender to allocate buffers for the exact message size and to release the buffer as soon as the RDMA operation is completed, without requiring the receiver to acknowledge the buffer as in MVAPICH. As a result, buffers for just a few messages are sufficient to satisfy the sender-side buffer needs. Second, the receiver-side buffers are also greatly reduced due to the fact that only outstanding messages are buffered and that a set of processes are sharing the same buffers. During program execution, not all channels reaches their peak usage at the same time. As a result, shared channels greatly reduce the total buffer memory usages. Third, the memory requirement for different programs varies significantly depending on how the MPI program is written. Some
programs such as Sweep3D and SMG2000 have many nonblocking small messages. As a result, its memory requirement for small message channels is much larger than other programs. This type of program also provides the most buffer memory saving opportunities for the proposed scheme with a more effective buffer memory management mechanism. For example, Sweep3D and SMG2000 on 1024 nodes, the scheme improves over MVAPICH in terms of maximum per-process memory by a factor of 18 and 15 respectively. Overall, my scheme improves over MVAPICH in terms of per-process maximum memory from 100% to 1846%. Moreover, if one considers the total memory used by the channels, which is reflected by $M_{\text{Ave:Shared}}$, there is almost another factor of 16 (the number of cores per node) reduction since the buffers for the shared channels are mostly the shared receiver-side buffers. In terms of the actual physical memory used for the channels (comparing the average per-process memory between the two schemes), the shared channels reduce the memory needed by a factor of 16 or a factor of 310. Another way to interpret the results is that if the system allocates the same amount of physical memory for both schemes, using the shared message channels it will be able to satisfy communications of small messages among a much larger number of processes.

### 4.3.2 Communication performance

The results in the previous sub-section show that the proposed shared message channels greatly reduce the memory requirement for small messages. Another important aspect of the design is the communication performance. Since the proposed shared channels use senders to manage the receiver-side buffers and require a mutex lock to ensure the consistency of the shared variables, there can be a performance penalty when multiple processes compete for the lock. To study the impact of this design on communication performance, the proposed techniques are implemented and incorporated in a prototype MPI library implementation, whose performance with a number of MPI programs is then measured.

The proposed small message channels for InfiniBand clusters are implemented
using the Verbs API [18]. The shared RDMA-based small message channels design is incorporated in a prototype MPI point-to-point communication systems (MYIB) that I previously developed [52]. The prototype library (MYIB) supports communications for both large and small messages with multiple rendezvous protocols for large message and the RDMA-based eager protocol for small message channels [24]. The newer system with the proposed small message channels can switch between using the traditional resource management scheme (MVAPICH small message channels) or the proposed one for managing small message channel resources. MYIB with the shared RDMA-based channel is referred to as MYIB++.

MYIB/MYIB++ supports five MPI point-to-point routines, MPI_Isend, MPI_Irecv, MPI_Send, MPI_Recv, and MPI_Wait, and as well as other maintenance routines including MPI_Init(), MPI_Finalize(), MPI_Comm_rank(), and MPI_Comm_size(). The prototype library can co-exist with MVAPICH [43]. When running an MPI application, I modify the five MPI point-to-point routine calls in the application to invoke my routines, all other MPI routines that are not supported fall back to MVAPICH. This system allows us to evaluate the performance of the proposed techniques.

The evaluation is performed on an InfiniBand cluster, Draco, that has 7 compute nodes with a total of 56 cores. Each node is a Dell Poweredge 1950 with two 2.33Ghz Quad-core Xeon E5345’s (8 cores per node) and 8GB memory. All nodes run Linux with the 2.6.9-42.ELsmp kernel. The compute nodes are connected by a 20Gbps InfiniBand DDR switch (CISCO SFS 7000D). The MPI library is MVAPICH2-1.2.rc1.

Seven benchmarks are used to investigate the potential performance impacts of the proposed techniques. Five programs are described in the previous sub-section: BT, CG, and SP from the NAS parallel benchmarks, Sweep3D from the ASCI benchmarks, and matrix multiply (MM). I was unable to make the prototype system work with SMG2000 and thus cannot provide results for SMG2000. For BT, CG, and SP, the CLASS C problem size is used in this experiment. For Sweep3d, the problem size is $240 \times 240 \times 200$ with $mni = 16, mk = 3$. The matrix multiply program multiplies two matrix of $8192 \times 8192$ doubles 100 times. The other two programs are Jacobi

42
and SparseMM. The Jacobi program uses Gauss-Siedel iterations to solve Laplace equations on a $8K \times 8K$ discretized unit square with Dirichlet boundary conditions. The SparseMM is a message passing implementation of the sparse SUMMA sparse matrix-matrix multiplication algorithm [4]. The program performs multiple times the self multiplication of a sparse matrix stored in file G3_circuit.mtx that is available in the University of Florida sparse matrix collection [13]. The matrix in G3_circuit.mtx is a sparse $1585478 \times 1585478$ matrix with 4623152 non-zero entries.

Table 4.2 shows the performance of the programs with three communication systems, MVAPICH, MYIB that has a small channel implementation similar to MVAPICH [24], and MYIB++ that has the proposed small message channels. The comparison should be made between MYIB and MYIB++: they have the same implementation for everything except the RDMA-based small message channels. The results of MVAPICH are listed in the table as a reference. For each communication scheme, the application total time and the communication time is given. Note that the small message channel is not only used to communicate small messages with the eager protocol, but also the control messages for communicating large message with the rendezvous protocol. Hence, only the total communication time in the program is considered. As can be seen from the table, the MYIB implementation is in general slightly better than MVAPICH mainly due to the simplified software architecture in MYIB and a better communication protocol for large messages. The results indicate that the MYIB implementation is at least comparable to the state of the art MPI library. Let us now compare the performance of MYIB and MYIB++. For most applications (CG, BT, SP, Jacobi, Sweep3D, and MM), MYIB++’s communication time is similar to that of MYIB: for all these cases, the communication times difference between the two schemes are within 5%. For Jacobi: with 16 processes, MYIB is sufficiently better and with 36 processes, MYIB++ is significantly better. Comparing MYIB with MYIB++, there are several factors that can affect the communication performance. On the one hand, MYIB++ (with the proposed techniques) has a more complex channel resource management scheme with a mutex lock to sequentialize accesses to
### Table 4.2: Communication performance

<table>
<thead>
<tr>
<th>Program</th>
<th>NPROCS</th>
<th>MVAPICH</th>
<th>MYIB</th>
<th>MYIB++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>total (sec.)</td>
<td>comm. (sec.)</td>
<td>total (sec.)</td>
</tr>
<tr>
<td>BT</td>
<td>16</td>
<td>316.12</td>
<td>13.35</td>
<td>316.50</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>138.48</td>
<td>9.78</td>
<td>137.38</td>
</tr>
<tr>
<td>CG</td>
<td>16</td>
<td>84.97</td>
<td>7.63</td>
<td>84.95</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>44.56</td>
<td>5.58</td>
<td>44.50</td>
</tr>
<tr>
<td>SP</td>
<td>16</td>
<td>587.29</td>
<td>25.67</td>
<td>587.37</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>237.42</td>
<td>17.16</td>
<td>243.97</td>
</tr>
<tr>
<td>SparseMM</td>
<td>16</td>
<td>15.61</td>
<td>12.01</td>
<td>11.44</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>14.35</td>
<td>11.92</td>
<td>11.61</td>
</tr>
<tr>
<td>Jacobi</td>
<td>16</td>
<td>271.45</td>
<td>3.11</td>
<td>273.15</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>130.45</td>
<td>10.41</td>
<td>130.56</td>
</tr>
<tr>
<td>Sweep3D</td>
<td>16</td>
<td>30.32</td>
<td>3.79</td>
<td>28.01</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>13.50</td>
<td>2.66</td>
<td>14.06</td>
</tr>
<tr>
<td>MM</td>
<td>16</td>
<td>29.69</td>
<td>20.55</td>
<td>30.33</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>15.38</td>
<td>11.38</td>
<td>15.43</td>
</tr>
</tbody>
</table>

### Table 4.3: Overhead of the mutex lock

<table>
<thead>
<tr>
<th>program</th>
<th>nprocs</th>
<th>total per node mutex time (sec.)</th>
<th>spins per comm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>16</td>
<td>0.0026</td>
<td>2.64</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>0.0065</td>
<td>1.02</td>
</tr>
<tr>
<td>CG</td>
<td>16</td>
<td>0.0010</td>
<td>1.40</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>0.0015</td>
<td>0.55</td>
</tr>
<tr>
<td>SP</td>
<td>16</td>
<td>0.0010</td>
<td>2.06</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>0.0030</td>
<td>1.17</td>
</tr>
<tr>
<td>SparseMM</td>
<td>16</td>
<td>0.0000</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>0.0001</td>
<td>0.35</td>
</tr>
<tr>
<td>Jacobi</td>
<td>16</td>
<td>0.0002</td>
<td>0.57</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>0.0003</td>
<td>0.16</td>
</tr>
<tr>
<td>Sweep3D</td>
<td>16</td>
<td>0.0008</td>
<td>1.62</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>0.0018</td>
<td>0.80</td>
</tr>
<tr>
<td>MM</td>
<td>16</td>
<td>0.0005</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>0.0013</td>
<td>0.50</td>
</tr>
</tbody>
</table>
the shared data structures. This results in increased communication overheads. On the other hand, MYIB++ uses much smaller message buffers with a significant faster sender-side buffer reclamation. This improves the cache performance for communications as the buffer access locality improves. The overall performance will depend on which factor dominates the time. I attribute MYIB++’s significantly better performance than MYIB to the sender side cache effect. Overall, this experiment shows that MYIB++ is not noticeably worse than MYIB. In fact, averaging across all the cases in the study, MYIB++ is 1.37\% better in communication time than MYIB. This indicates that the proposed scheme does not introduce notable overheads in comparison to the traditional scheme.

Table 4.3 shows the overhead introduced by the mutex lock: 8 processes in the sender may compete for the mutex lock when all try to communicate at the same time. The table shows the total time spent in the spin lock waiting for the lock to be free and the average number of spins for each communication. As can be seen from the table, the time spent on waiting is negligible in comparison to the communication time. Moreover, the average number of spins per communication is mostly less than 3, which indicates that processes almost do not wait for the lock to be freed for the communication. As previously mentioned, the critical section protected by the lock is very small: only a constant number ($O(1)$) of instructions to obtain a buffer from the available buffer poll and updates a few pointers. With 8 processes, the overhead for competing for the lock is negligible. Note that in cases when the number of cores in an SMP node is large (e.g., 128). The overhead may be significant. In this case, a subset of the processes within an SMP node may share a channel to minimize the overhead.

### 4.4 Chapter Summary

This work attempts to improve upon the existing design for RDMA-based MPI small message channel implementations in order to overcome its limitations, which include high memory requirements and scalability issues. Two techniques are explored:
(1) a method for eliminating the need for a resource-intensive design component known as persistent buffer association from RDMA-based small message channel design and (2) a shared RDMA-based small message channel design that is built upon the previous technique and allows multiple MPI processes within an SMP node to share small message channels. The first technique improves small message channel resource utilization by reducing the sender-side fragmentation and eliminating the dedication of buffer memory upon channel initialization. This technique also severs the per-process association of sender-side and receiver-side buffers enabling greater flexibility in buffer memory management. The second technique leverages this flexibility to design a scheme that further improves the memory requirement and scalability, reducing the number of small message channels needed to run an MPI application from $O(n^2)$ where $n$ is the number of MPI processes to $O(N^2)$ where $N$ is the number of nodes for the application. In addition to lowering the number of small message channels, the shared channel scheme also reduces buffer fragmentation on the receiver-side and markedly reduces the average number of receive buffers needed per-process. The memory analysis results show that in some cases these techniques achieve over a hundredfold reduction in the average per-process memory requirement and a nearly tenfold reduction in the maximum memory requirement in comparison to the existing system. The communication performance results show that this novel design achieves these improvements without adding noticeable overheads or sacrificing the performance benefits of RDMA.

The techniques developed in this work are implemented for Infiniband networks using the Verbs library. However, the techniques are generally applicable to any interconnect that supports the in-order delivery of message data associated with RDMA operations.
CHAPTER 5

NEAR-OPTIMAL RENDEZVOUS PROTOCOLS

Achieving high performance communication for large messages on RDMA-enabled clusters is challenging mainly for two related reasons. The first is the protocol complexity. As alluded to in Section 1.3, copying data can introduce significant overheads and should in general be avoided for large communications. Hence, all existing large message protocols are rendezvous protocols with multiple rounds of control messages, which can result in various problems such as unnecessary synchronizations and communication progress issues [10, 45, 47, 52, 54]. The second is the complexity in the protocol invocation scenario. MPI allows both the sender and the receiver to mark the times when a communication can start (e.g. MPI_Isend/MPI_Irecv) and when a communication must be completed (e.g. MPI_Wait). There are many combinations of the relative timing of these events, which can greatly affect the performance of a given protocol. I use the term protocol invocation scenario to denote the timing of the events in a communication. It is virtually impossible to design one scheme that guarantees high performance for all cases.

In this chapter, I investigate a profile-driven, compiler-assisted protocol customization approach to maximize the performance for communicating large messages. Instead of using the same protocol for any protocol invocation scenario, this approach attempts to use fine-grained analysis of the protocol invocation scenario to improve protocol design. It begins by identifying the protocol invocation scenario for each critical communication through analyzing the execution traces of an MPI program.
(program profile data) and/or by analyzing the program, and then choosing the most appropriate protocol for each scenario. Note that it might not be necessary to apply protocol customization for all communication routines in an application. Customizing the protocols in a small number of critical communication routines may yield significant improvement.

In order for profile-driven, compiler-assisted protocol customization to be effective, (1) trace analysis and/or compiler analysis techniques must be developed to accurately determine protocol invocation scenarios; and (2) efficient communication protocols must be designed for all invocation scenarios. This research focuses on the second item: obtaining efficient communication protocols. I analyze existing protocols for communicating large messages on RDMA-enabled clusters and show that they are not ideal in many situations and then develop a set of six protocols that can deliver near-optimal performance for all protocol invocation scenarios by leveraging RDMA capabilities: when the protocol invocation scenario can be decided for a communication, one of the six protocols can be selected by the compiler or runtime system to achieve high performance. Finally, I implement all of the proposed protocols on InfiniBand and evaluate the potential benefits of protocol customization using micro-benchmarks and application benchmarks. The results indicate that protocol customization using fine-grained protocol invocation scenario information can significantly improve MPI communication performance.

5.1 Protocol invocation scenarios

Each rendezvous protocol requires the exchange of one or more control messages. The control messages introduce implicit synchronizations between the sender and the receiver: when a protocol requires that a party $P_1$ respond to a control message from the other side, $P_1$ cannot make progress unless the other party has sent this control message. Due to such implicit synchronizations, the performance of a rendezvous protocol can be significantly affected by the protocol invocation scenario, i.e., the timing of the communication related events in the sender and the receiver. Next,
I will discuss protocol invocation scenarios and then show why existing rendezvous protocols cannot deliver high performance in many situations.

There are four critical events in each MPI point-to-point communication: (1) the time when the sender can start the communication, which corresponds to the \texttt{MPI\_Isend} call at the sender side and will be denoted as $SS$, (2) the time when the sender must complete the communication, which corresponds to the \texttt{MPI\_Wait} call at the sender side and will be denoted as $SW$, (3) the time when the receiver can start the communication, which corresponds to the \texttt{MPI\_Irecv} call at the receiver side and will be denoted as $RS$, and (4) the time when the receiver must complete the communication, which corresponds to the \texttt{MPI\_Wait} at the receiver side and will be denoted as $RW$.

I will use the notations $SS$, $SW$, $RS$, and $RW$ to denote both the events and the timing of the events. The sender may or may not have computations between $SS$ and $SW$; and similarly, the receiver may or may not have computation between $RS$ and $RW$. When there are computations at those points, it is desirable to overlap the communication with these computations. After $SW$, the sender is blocked and does not perform any useful work until the communication is completed at the sender side. Similar, after $RW$, the receiver is blocked and does not perform any useful work until the communication is completed at the receiver side.

Let $A, B \in \{SS, SW, RS, RW\}$. The notion $A \leq B$ will be used to denote that event $A$ happens before or at the same time as event $B$, $A = B$ to denote that event $A$ happens at the same time as event $B$, and $A < B$ to denote that $A$ happens before $B$. Ordering events in one process is trivial: clearly, we have $SS \leq SW$ and $RS \leq RW$. Note that $SS$ and $SW$ happens at the same time in a blocking send call (\texttt{MPI\_Send}); and $RS$ and $RW$ happens at the same time in a blocking receive call (\texttt{MPI\_Recv}). For events in two processes, the order is defined as follows. Let event $A$ happens in process $P_A$, and event $B$ happens in process $P_B$. $A < B$ if after $A$, $P_A$ has time to deliver a control message to $P_B$ before $B$. $A = B$ denotes the case when each party does not have time to deliver a control message to other party before the
event in that party happens. This method of event ordering is similar to that found in the dynamic protocol selection scheme discussed in the previous chapter. Figure 5.1 shows the ordering of events in two processes.

Since \( SS \leq SW \) and \( RS \leq RW \), there are only six different orderings among the four events in a communication: \( SS \leq SW \leq RS \leq RW \), \( SS \leq RS \leq RW \leq SW \), \( RS \leq RW \leq SS \leq SW \), \( RS \leq SS \leq RW \leq SW \), and \( RS \leq SS \leq SW \leq RW \). However, the ordering of the communication events is not the only factor that affects protocol design, the actual timing of the events also has an impact as will be shown in the following sections.

### 5.2 Existing rendezvous protocols and their limitations

There are three existing rendezvous protocols developed for RDMA-enabled systems, the traditional sender-initiated RDMA write-based protocol [24], the sender-initiated RDMA read-based protocol [54], and the receiver-initiated protocol [45, 47]. I will briefly introduce each protocol and discuss their limitations. Ideally, in a rendezvous protocol, when both sender and receiver are ready for the communication, that is, both \( SS \) and \( RS \) happen, data transfer should start to maximize the overlap with the computations between \( SS \) and \( SW \) in the sender side and between \( RS \) and \( RW \) in the receiver side. None of these protocols can achieve this in all cases.

Examples of the sender-initiated RDMA write-based rendezvous protocol [24] are shown in Figure 5.2. In this protocol, the sender initiates the communication by
sending a **SENDER READY** packet, the receiver then responds with a **RECEIVER READY** packet, after that the message data are RDMA written and a FIN packet is sent to indicate the completion of the communication. The sender must then wait for the communication of the data to complete before it can return from the operation. The examples illustrated in Figure 5.2a and 5.2b demonstrate cases where the sender-initiated RDMA write-based rendezvous protocol delivers sub-optimal performance. In Figure 5.2a, $SS = RS$ and since both parties for the rendezvous communication has arrived, ideally, data transfer should start within one control message time of $SS$ and overlaps with the computation between $SS$ and $SW$ and between $RS$ and $RW$. However, using this protocol, data transfer happens after $RW$ with no communication-computation overlap (sender has been idling at $SW$, waiting to complete the protocol). In Figure 5.2b, $SS < RS$ and the receiver responds to the **SENDER READY** message right away. Again, ideally, data transfer should happen when both $SS$ and $RS$ happen (both sides are ready for the communication). However, since sender is in the computation when **RECEIVER READY** arrives, the data transfer happens at a later time in $SW$: there is no overlap between computation and communication. Notice that the performance penalties for the inefficient protocol depend on the program structure: for Figure 5.2a, the sender can idle in $SW$ for a very long time depending on the amount of computation between $RS$ and $RW$; for Figure 5.2b, the receiver can idle in $RW$ for a very long time depending on the amount of computation between $SS$ and $SW$.

Examples of the sender-initiated RDMA read-based protocol [54] are shown in Figure 5.3. In this protocol, the receiver responds to the **SENDER READY** packet with an RDMA read operation. After the RDMA read operation is completed, the receiver sends a FIN packet to the sender and completes the operation. The sender exits the operation after it receives the FIN packet. In comparison to the RDMA write-based protocol, this protocol eliminates the **RECEIVER READY** message, which may result in better communication progress [54]. However, this protocol also suffers from some limitations as shown in Figure 5.3. In Figure 5.3a, $SS = RS$. Using this protocol,
data transfer happens at $RW$, which is not ideal. In Figure 5.3b, $RS < SS$. With this protocol, the receiver does nothing at $RS$ and data transfer still happens at $RW$.

An example of the receiver-initiated protocol [45] is shown in Figure 5.4. In this protocol, the sender does nothing at $SS$ if $SS < RS$. The receiver sends a $RECEIVER\_READY$ packet to the sender, which carries the receiving buffer information. When the sender gets this packet, it can directly deposit the data message into the receiver user space. As shown in Figure 5.4, when $SS = RS$, the protocol is not ideal as the data transfer starts at $SW$.

There are other cases that all existing protocols can only give sub-optimal performance. Since none of the protocols are ideal in some cases (e.g. when $RS = SS$), the schemes [47, 51] that combine the sender-initiated protocol with the receiver-initiated protocol are also not ideal in such cases. Hence, to effectively support profile-driven compiler-assisted protocol customization, new efficient protocols must be developed for the scenarios that existing protocols cannot perform well.
5.3 Near-optimal communication protocols for large messages

In this section, protocols are developed for communicating large messages that can deliver near-optimal performance for all protocol invocation scenarios. The following assumptions are made:

- Data transfer cannot start unless both SS and RS happen. This is typical for sending large messages: both sides must be ready for the data to be communicated.

- The delay (cost) associated with sending and receiving a control message is negligible.

- RDMA read and RDMA write have similar performance.

- The sender can buffer the data message when necessary. Buffering at the sender side, even for large messages, is practical since it does not require the excessive per-pair buffers. However, buffering requires CPU time and thus, must be used
with care. Hence, it is further assumed that buffering at the sender can only be performed when the sender is blocked.

Let $REND$ be the time when both $SS$ and $RS$ happen (the rendezvous time of the communication), $comm(msg)$ be the time to transfer the message with either RDMA write or RDMA read, and $copy(msg)$ be the time to make a local copy of the message. Under the above assumptions, an ideal communication scheme should have the following properties.

- It should start the data transfer at the earliest time, which is $REND$. Starting the data transfer at the earliest time also maximizes communication-computation overlaps. It follows that the receiver should complete the operation at $REND + comm(msg)$.

- When $REND \leq SW$, the sender should send the message at $REND$ and complete the operation at $REND + comm(msg)$.

- When $SW < REND$, the sender can buffer the data, use a control message to notify the receiver about the buffer, and return from the operation. The receiver can get the data from the buffer; and the buffer can be released in a later communication operation after the receiver gets the data. Thus, in this case, the sender should complete the operation at $SW + copy(msg)$.
Note that this ideal communication scheme may not be optimal in that the communication completion times for the sender and the receiver may not be the earliest possible times in all cases. For example, the concurrency of sending data and copying data simultaneously is not considered. Improvements can be made by exploiting such concurrency, as will be shown in one of the proposed protocols. Hence, I will say that this ideal scheme is near-optimal. The proposed protocols are near-optimal in the sense that, ignoring the control message overheads, they have the same communication start times and completion times as the ideal communication scheme described above.

Next, protocols for all protocol invocation scenarios are presented. All protocol invocation scenarios can be grouped into three classes: \( SS < RS \), \( SS = RS \), and \( RS < SS \). For a \( SS < RS \) scenario, the sender arrives at the communication earlier than the receiver: the sender can notify the receiver that it is ready for the communication at \( SS \) and the receiver can get the notification at \( RS \). Similarly, for a \( RS < SS \) scenario, the receiver arrives at the communication earlier than the sender: the receiver can notify the sender that it is ready for the communication at \( RS \) and the sender can get the notification at \( SS \). For a \( SS = RS \) scenario, the sender and the receiver arrive at the communication at similar times: \( SS \) and \( RS \) are within one control message time.

Let us first consider the scenarios with \( SS < RS \). The scenarios in this class are further partitioned into three cases with each case having a different protocol. The three cases are: \( SS \leq SW < SW + copy(msg) < RS \leq RW \), \( SS \leq SW < RS(\leq SW + copy(msg)) \leq RW \), and \( SS < RS \leq \{SW \text{ and } RW\} \). Here, \( SW + copy(msg) \) is \( copy(msg) \) time after \( SW \). In the case \( SS < RS \leq \{SW \text{ and } RW\} \), \( SW \) and \( RW \) both happen no earlier than \( RS \) and the order between \( SW \) and \( RW \) does not matter.

Figure 5.5a shows the scenario for \( SS \leq SW < SW + copy(msg) < RS \leq RW \), where \( SW \) is much earlier than \( RS \). The proposed protocol for this case, shown in Figure 5.5b, is called the \( copy-get \) protocol. In this protocol, the sender copies
the message data to a local buffer at $SW$ (this has no costs since sender is blocked for the communication and cannot do anything useful). After the data are copied, the sender issues a READY message, which contains the address of the local buffer and other related information to facilitate the RDMA read from the receiver, to the receiver. The task in the sender side is completed; and the sender can exit the operation. When the receiver gets the READY message, it performs an RDMA read to obtain the data from the sender buffer. The sender side buffer must be released at some point. Since this is a library buffer that the application will not access, the information for the receiver to notify the sender that the buffer can be released can be piggybacked in a later control message. The copy_get protocol leverages the RDMA capability and allows the sender to complete the communication before the receiver even arrives. For the sender, the operation completes at $SW + \text{copy}(msg)$. The data transfer starts at $REND = RS$ and the receiver completes the communication at $REND + \text{comm}(msg)$. Hence, this protocol is near-optimal for both the sender and the receiver.

Figure 5.6a shows the scenario for $SS \leq SW < SW + \text{copy}(msg) < RS \leq RW$, where sender blocks ($SW$) slightly earlier than receiver arriving at the communication ($RS$) with not enough time to copy the whole message. The proposed protocol
for this case, shown in Figure 5.6b, is called the copy_check_put protocol. In this protocol, the sender sends a SENDER_READY message to receiver at SS. At SW, the sender starts copying the message data to a local buffer while monitoring control messages from the receiver. This can be implemented by repeatedly copying a small chunk of data and checking the message queue. Like in the copy_get protocol, these operations have no costs since the sender is blocked for the communication and cannot do anything useful. When the receiver arrives at RS, it will receive the SENDER_READY and send a RECEIVER_READY message, which should arrive at the sender before the sender finishes making a local copy. When the sender gets the RECEIVER_READY message, it sends partial data to the receiver while continuing to copy the message concurrently. We will assume that the system knows the copy and data transmission speeds and can determine the amount of data to be copied and to be transferred so that the combination of copied data and transferred data covers the whole message and that the (partial) data copy and (partial) data transfer complete at the same time. Note that this assumption can be approximated in practice. After that, the sender initiates the sending of the copied data and the FIN packet, and then returns from the communication. The copied data will be released in a later communication operation. For this protocol, the sender completes the operation before SW + copy(msg) since
it returns after the message is partially copied (initiating a communication does not take time). This is due to the concurrent sending and copying data in the protocol. The data transfer starts at $REND = RS$ and the receiver completes the operation at $REND + comm(msg)$. Hence, this protocol is near-optimal.

For $SS < RS \leq \{SW \text{ and } RW\}$ scenarios, the traditional sender-initiated RDMA read-based protocol is near-optimal. Using this protocol, data transfer starts at $REND = RS$; and both the sender and the receiver complete the communication at $REND + comm(msg)$.

Let us now consider the second class: $SS = RS$. This is one case when all existing rendezvous protocols are not ideal. However, if trace/profile/static analysis can draw the conclusion that $SS$ and $RS$ are within one control message time, the solution is straight-forward: waiting for the corresponding control message at $SS$ or $RS$.

We will call such protocols delayed sender-initiated protocols and delayed receiver-initiated protocols. Figure 5.7a shows a delayed sender-initiated RDMA read-based protocol. In this protocol, the receiver adds a delay time $RS$ (marked as $RS(begin)$ and $RS(end)$ in Figure 5.7a). During the delay, the receiver repeatedly pulls the control message queue waiting for the $SENDER_{READY}$ message to arrive at $RS$ so that

![Diagram of delayed rendezvous protocols](image)

(a) Delayed sender-initiated protocol    (b) Delayed receiver-initiated protocol

Figure 5.7: Delayed rendezvous protocols for $SS = RS$ scenarios
data transfer can start before the receiver leaves $RS$. Notice that the delay is less than one control message time and the communication starts within one control message time from $REND$. Hence, both the sender and receiver will complete the operation at $REND + \text{comm}(msg)$. Figure 5.7b shows the delayed receiver-initiated protocol where the delay is added to the sender at $SS$.

Finally, for the third class where receiver arrives earlier than the sender ($RS < SS$), the traditional receiver-initiated protocol, as shown in Figure 5.4, can achieve near-optimal performance. Data transfer starts exactly at $SS = REND$, which is the earliest time possible. Both sender and receiver will complete the operation at $REND + \text{comm}(msg)$, which is the same as the ideal communication scheme. Notice that when the receiver arrives at $RS$ and $RW$ much earlier than the sender, it must wait for the sender in order to complete the communication: there is an inherent synchronization from the sender to the receiver in every communication. As a result, the case when receiver arrives much earlier cannot be further optimized.

The six protocols that are discussed in this section, the copy get protocol, the copy check put protocol, the sender-initiated RDMA read-based protocol, the delayed sender-initiated protocol, the delayed receiver-initiated protocol, and the receiver initiated protocol, should be able to achieve near-optimal performance for communicating large messages in any protocol invocation scenario. By combining these protocols with a profile driven static analysis scheme that identifies protocol invocation scenarios, protocol customization can potentially achieve near-optimal communication performance for all situations.

### 5.4 Performance study

In this section, I evaluate the performance of the proposed rendezvous protocols and study the potential benefits of protocol customization with micro-benchmarks and application benchmarks. I have implemented the six protocols discussed in the previous section over InfiniBand using the Verbs API [18] in six versions of `MPI_Isend`,
### Figure 5.8: Progress micro-benchmark

<table>
<thead>
<tr>
<th>Process 1:</th>
<th>Process 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>Loop:</td>
</tr>
<tr>
<td>barrier()</td>
<td>barrier()</td>
</tr>
<tr>
<td>comp1</td>
<td>comp4</td>
</tr>
<tr>
<td>MPI_Isend()</td>
<td>MPI_Irecv()</td>
</tr>
<tr>
<td>comp2</td>
<td>comp5</td>
</tr>
<tr>
<td>MPI_Wait()</td>
<td>MPI_Wait()</td>
</tr>
<tr>
<td>comp3</td>
<td>comp6</td>
</tr>
</tbody>
</table>

**MPI_Irecv, MPI_Send, MPI_Recv, and MPI_Wait.** My library can co-exist with MVAPICH. MPI functions that are not supported by our library can be realized by MVAPICH. This allows the performance of MPI programs to be compared with point-to-point routines using our protocols to that with MVAPICH.

The evaluation is performed on an InfiniBand cluster that has 16 compute nodes with a total of 128 cores. Each node is a Dell Poweredge 1950 with two 2.33Ghz Quad-core Xeon E5345’s (8 cores per node) and 8GB memory. All nodes run Linux with the 2.6.9-42.ELsmp kernel. The compute nodes are connected by a 20Gbps InfiniBand DDR switch (CISCO SFS 7000D). The performance of protocol customization is compared with that of the default MVAPICH2-1.2.rc1, which uses the sender-initiated RDMA write-based protocol to communicate large messages.

### 5.4.1 Micro-benchmark results

The benchmark shown in Figure 5.8, called progress benchmark, is used to evaluate the communication progress and communication-computation overlap capability of the proposed scheme. In this benchmark, the time for 1000 iterations of the loop is measured. Inside the loop, a barrier is first called to synchronize the sender and the receiver. After that, the sender performs some computation `comp1`, calls `MPI_Isend` to start the send operation, performs some more computation `comp2`, calls `MPI_Wait` to complete the send operation, and performs some more computation `comp3`. Similarly, the receiver also performs some computation `comp4` after the barrier, calls `MPI_Irecv` to start the receive operation, performs some more
computation \( \text{comp}5 \), calls \( \text{MPI\_Wait} \) to complete the receive operation, and performs some more computation \( \text{comp}6 \). The message size and the computation in between the communication routines are parameters. The notation configuration \((\text{comp}1, \text{comp}2, \text{comp}3, \text{comp}4, \text{comp}5, \text{comp}6)\) will be used to represent the configuration of the benchmark, where \( \text{comp}X \) represents the duration of the computation (in the unit of a basic loop). For each computation, the larger the number is, the longer the computation lasts. In the discussion, I will use notation \( C(\text{comp}X) \) for the time for \( \text{comp}X \) computations and \( T(\text{msize}) \) for the time to transfer a message of \( \text{msize} \) bytes. In the experiment, \( C(X) + C(Y) \approx C(X + Y), C(1) \approx 18\mu s \) and \( T(100KB) \approx 90\mu s \).

This benchmark can be configured to test the communication progress and the communication-computation overlap capability under various conditions. Four experiments are designed to show the advantages of the integrated protocols, in particular, the advantages of the hybrid protocol and the receiver-initiated protocol. Since the performance when both sender and receiver arrive at similar times is implied in the pingpong results, I will focus on the cases when either the sender or the receiver arrives early.

I perform experiments using this micro-benchmark with different orderings of events and different relative timings. Protocol customization consistently achieves high performance. Let us now consider the results for three representative cases. The first case has configuration \((1, 1, 48, 30, 19, 1)\), which emulates the case when \( SS \) and \( SW \) are much earlier than \( RS \) and \( RW \) as shown in Figure 5.5a. The second case has configuration \((10, 30, 10, 10, 30, 10)\), which emulates the case when \( SS = RS \) as shown in Figure 5.2a. The third case has configuration \((10, 20, 20, 1, 40, 9)\), which emulates the case when \( RS < SS \) as shown in Figure 5.3b. Note that for all cases, both sender and receiver have a total of 50 units of computations, which translate to roughly \( C(50) = 50 \times 18 = 900 \mu s \) if both sides perform the computation concurrently.

Results for configuration \((1, 1, 48, 30, 19, 1)\) with different message sizes are shown in Figure 5.9a. Using the default rendezvous protocol in MVAPICH, there is an
implicit synchronization from $RS$ to $SW$, which results in the computation before $RS$ (30 units) at the receiver and the computation after $SW$ (48 units) at the sender to be sequentialized. Hence, the total time for each iteration is roughly $T(msize) + C(30 + 48)$. On the other hand, with protocol customization, the most effective protocol is the copy_get protocol in Figure 5.5b, where the sender makes a local copy of the buffer and leaves the communication. With this protocol, the total time for each iteration is roughly $\text{copy}(msize) + C(50)$, which is much better than the result with the default MVAPICH as shown in Figure 5.9a. Notice that this is one scenario where no rendezvous protocol can perform well: copy_get is not a true rendezvous protocol since the sender leaves the communication before the receiver starts the communication. Notice also that copying data introduces significant overheads as shown in the upward slope for the curve for our scheme in Figure 5.9a.

Results for configuration $(10, 30, 10, 10, 30, 10)$ with different message sizes are shown in Figure 5.9b. This is the case when the READY messages from both sender and receiver pass each other and no existing protocol is ideal as discussed in Section 5.2. With the default MVAPICH protocol, data transfer is performed at $SW$ (and $RW$), and no communication-computation overlap is achieved. The per iteration time is thus roughly $T(msize) + C(50)$: the time increases linearly with the message size as shown in Figure 5.9b. The most effective protocol for this situation...
is the delayed receiver-initiated protocol shown in Figure 5.7b, where the sender repeatedly polls the incoming message queue for the \texttt{RECEIVER\_READY} message. Using this protocol, the communication can be completely overlapped with computations between $SS$ and $SW$ at the sender side and $RS$ and $RW$ at the receiver side; and the per iteration time is roughly $C(50)$, shown as a flat line in Figure 5.9b.

Results for configuration (10, 20, 20, 1, 40, 9) with different message sizes are shown in Figure 5.10. This case emulates the situation at Figure 5.3b. With the default protocol, the communication starts at $RW$. Hence, the per iteration is roughly $C(41) + T(msize) + C(20) = C(61) + T(msize)$. Using the near-optimal receiver initiated protocol, the communication is overlapped completely with computation and the total time is roughly $C(50)$.

These results demonstrate that by using near-optimal protocols for different protocol invocation scenarios, protocol customization avoids the performance penalties due to the mismatch between the protocol and the protocol invocation scenarios and can achieve higher performance in comparison to traditional rendezvous protocols in many cases. Moreover, the improvement from protocol customization depends not only on the system communication performance, but also on program structures.
I use five application benchmarks to investigate the potential performance benefits of protocol customization. Three benchmarks are from the NAS parallel benchmarks [44]: BT, CG, and SP. We use the CLASS C problem size for all of the three programs. The other two programs are jacobi and sparsemm. The jacobi program uses Gauss-Siedel iterations to solve Laplace equations on a $8K \times 8K$ discretized unit square with Dirichlet boundary conditions. The sparsemm is a message passing implementation of the sparse SUMMA sparse matrix-matrix multiplication algorithm [4]. The program performs multiple times the self multiplication of a sparse matrix stored in file G3_circuit.mtx that is available in the University of Florida sparse matrix collection [13]. The matrix in G3_circuit.mtx is a sparse $1585478 \times 1585478$ matrix with $4623152$ non-zero entries.

Since I have not developed trace and compiler analysis techniques to identify the most effective protocol for each communication, it is not possible to thoroughly evaluate the benefit of protocol customization. The methods used to select protocols in this experiment, which will be described next, are preliminary and may not select the most efficient protocols for all communications. Hence, the results in this section only represent the lower bound of the potential improvement achievable through protocol customization. In the experiments, the following is done. MPI program execution traces are manually examined, which gives the timing of each MPI routine calls, and use the timing information to decide the protocol for each communication. This approach is not always effective since the relative timing of critical communication events may change when the communication protocol is changed. Developing more sophisticated techniques that can select protocols more effectively is left for future research. In addition to the trace-driven protocol customization, we also run each individual protocol for each of the programs. The reported performance results for protocol customization are the best communication times from both the trace-driven execution and the individual protocol execution.

Table 5.1 shows the total application times, total communication times, and the
Table 5.1: Performance on 16 processes (one process per node)

<table>
<thead>
<tr>
<th></th>
<th>MVAPICH</th>
<th></th>
<th>Customization</th>
<th></th>
<th>Comm. improv. percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total (sec.)</td>
<td>comm. (sec.)</td>
<td>total (sec.)</td>
<td>comm. (sec.)</td>
<td></td>
</tr>
<tr>
<td>BT</td>
<td>321.76</td>
<td>10.11</td>
<td>215.55</td>
<td>3.65</td>
<td>177.0%</td>
</tr>
<tr>
<td>CG</td>
<td>35.66</td>
<td>3.41</td>
<td>35.12</td>
<td>3.02</td>
<td>12.9%</td>
</tr>
<tr>
<td>SP</td>
<td>180.57</td>
<td>6.53</td>
<td>176.94</td>
<td>2.78</td>
<td>134.9%</td>
</tr>
<tr>
<td>sparsemm</td>
<td>16.35</td>
<td>12.75</td>
<td>10.51</td>
<td>6.92</td>
<td>84.2%</td>
</tr>
<tr>
<td>jacob</td>
<td>282.94</td>
<td>2.88</td>
<td>282.33</td>
<td>2.35</td>
<td>22.6%</td>
</tr>
</tbody>
</table>

Table 5.2: Performance on 121/128 processes (8 processes per node)

<table>
<thead>
<tr>
<th></th>
<th>MVAPICH</th>
<th></th>
<th>Customization</th>
<th></th>
<th>Comm. improv. percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total (sec.)</td>
<td>comm. (sec.)</td>
<td>total (sec.)</td>
<td>comm. (sec.)</td>
<td></td>
</tr>
<tr>
<td>BT</td>
<td>99.90</td>
<td>21.00</td>
<td>98.52</td>
<td>16.45</td>
<td>27.7%</td>
</tr>
<tr>
<td>CG</td>
<td>16.45</td>
<td>6.87</td>
<td>16.16</td>
<td>6.60</td>
<td>4.1%</td>
</tr>
<tr>
<td>SP</td>
<td>56.28</td>
<td>13.77</td>
<td>55.70</td>
<td>12.12</td>
<td>13.6%</td>
</tr>
<tr>
<td>sparsemm</td>
<td>14.06</td>
<td>12.44</td>
<td>11.97</td>
<td>10.56</td>
<td>17.8%</td>
</tr>
<tr>
<td>jacob</td>
<td>97.15</td>
<td>63.78</td>
<td>92.60</td>
<td>58.95</td>
<td>8.2%</td>
</tr>
</tbody>
</table>
communication improvement percentages using the proposed protocol customization scheme over MVAPICH for the programs running on 16 processes (one process per node). The communication time includes all Send, Isend, Recv, Irecv, and Wait times, which account for the majority of all communication times in these benchmarks. As can be seen from the table, protocol customization achieves significant improvement over MVAPICH for all the programs. The reason that protocol customization provides better performance for different programs are different. For BT, SP, and jacobi, the main reason is that protocol customization can explore the communication and computation overlapping opportunities better than the traditional protocol. For sparsemm, the main reason is the use of the copy-get protocol that eliminates the unnecessary synchronization from the sender to the receiver: the computation load is not balanced in this sparse matrix-matrix multiplication program and unnecessary synchronizations can introduce large waiting time, which is reduced with protocol customization. For CG, the performance gain is mainly from using a simpler receiver initiated protocol to carry out the communication. As can be seen from the table, although communication does not account for a large percentage of the total application time in BT, CG, SP, and jacobi, the improvement in communication times transfers into improvement of the total application time. For sparsemm, the total application time is also significantly improved since the communication time dominates this program.

Table 5.2 shows the results for the programs running on 121/128 processes (121 for BT and SP, 128 for CG, sparsemm, and jacobi) with 8 processes running on each node. One of the main difference between running one process per node and 8 processes per node is the intra-node communication. Since intra-node communication does not use the rendezvous protocol, our protocol customization is only applied to a portion of all communications (inter-node communications with large messages) in this experiment. This is the main reason that the communication improvement percentage is much lower for the 121/128 processes cases. However, as shown in the table, having a better inter-node communication mechanism with protocol customization still
provides noticeable improvement for all the benchmarks.

## 5.5 Chapter Summary

This work develops a foundation for protocol customization through the design of a set of rendezvous protocols that can collectively provide near-optimal performance for all possible protocol invocation scenarios. I create an empirical framework for analyzing communication performance in relation to key timings associated with the invocation of rendezvous protocols. Using this framework, I categorize six protocol invocation scenarios that cover all possible communication timings and map three such scenarios to existing rendezvous protocols providing near-optimal performance. Additionally, I present three new rendezvous protocol designs that provide near-optimal performance each of the remaining scenarios. The performance evaluation shows that these protocols improve protocol progress and can achieve an improvement in communication performance of as much as 177%. Overall, this work demonstrates that the flexibility of RDMA can be leveraged to develop protocols customized to different situations and that protocol customization has the potential to provide significant improvement for MPI communication.
CHAPTER 6

DYNAMIC PROTOCOL SELECTION

The previous chapter establishes that a primary limitation of traditional rendezvous protocols is their static nature and protocol customization, which tailors rendezvous protocol design to specific protocol invocation scenarios, can significantly improve MPI communication performance. However, the profile-driven and/or compiler-assisted approach previously discussed is not always a practical way to apply MPI optimization. For example, this type of optimization is not suited to MPI applications that will not run many times since it requires a program to be run at least twice, once to collect profile data and once to apply the optimization, and the cost of the profiling run cannot be amortized over a large number of executions. Additionally, some MPI applications have data-driven communication where the communication pattern is dependent on the dataset accessed by the MPI application. In this case, the profile data would only be valid for the dataset used in the profile run, which is not useful. In addition, using profile-driven and/or compiler-assisted based approach also presents additional hurdles for MPI users: only advanced users will be able to use those techniques. This work aims at supporting protocol customization without any modification to MPI source code or the way MPI programs are executed.

A scheme is developed that applies protocol customization at run-time, without relying on profile data, by integrating several protocols into a single system that dynamically selects an optimized protocol for each communication. The challenges with developing such a scheme include (1) gathering useful protocol invocation scenario information at run-time without adding to protocol complexity, and (2) integrat-
ing multiple protocols into a single dynamic scheme without introducing excessive overheads and while continuing to support full MPI semantics. To address the first challenge, I note that the analysis in the previous chapter shows process arrival (i.e. the order of $SS$ and $RS$) by itself to be a strong indicator for selecting an optimized rendezvous protocol. An example of this is that the receiver-initiated protocol is always near-optimal when the receiver arrives first ($RS < SS$). Thus, to simplify the dynamic collection of protocol invocation data, my scheme only attempts to determine the ordering of process arrival to the communication and use this information, along with message size, to make protocol selection decisions. To address the second challenge, I develop protocol integration techniques to deal with issues such as perceived protocol mismatches between the sender and receiver, matching control messages and supporting MPI wildcards. By dynamically adapting with multiple protocols, my integrated system is much more robust and efficient in dealing with different protocol invocation scenarios than traditional MPI libraries that utilize one fixed rendezvous protocol.

The remainder of this chapter is organized as follows. I first present the proposed dynamic selection scheme, which is a seamless integration of the eager protocol for small messages, the hybrid protocol for medium-sized messages, and the sender-initiated and receiver-initiated protocols for large messages. I then describe techniques related to the integration of the protocols. Finally, I present experimental results obtained by implementing the proposed scheme for InfiniBand clusters and comparing its performance to traditional schemes, which shows that in comparison to traditional statically selected protocol schemes, the proposed dynamic protocol customization scheme reduces unnecessary synchronizations, decreases the number of control messages that are in the critical path of communications, and improves the communication progress, which results in a better communication-computation overlap capability.
Figure 6.1: The timing when the sender and receiver arrive at a communication

6.1 The Dynamic Selection Scheme

In the proposed scheme, each party (sender or receiver) tries to notify the other party of its readiness for the communication as soon as it arrives at the operation. The protocol for a given communication is selected not only based on the message size, but also based on the timing when the sender and the receiver arrive at the communication. There are three cases, shown in Figure 6.1, that characterize the timing when the parties arrive at the communication:

- **Sender arrives early** (Figure 6.1a). In this case, after the sender arrives at the communication, it can send a control message to the receiver and the message will be delivered before the receiver arrives at the communication.

- **Receiver arrives early** (Figure 6.1b). In this case, after the receiver arrives at the communication, it can send a control message to the sender and the message will be delivered before the sender arrives at the communication.

- **Both arrive at similar times** (Figure 6.1c). This is the case when each party does not have time to deliver a control message before the other party arrives.

I refine the traditional rendezvous protocol with three protocols that are customized to different situations, the hybrid protocol (*HYBRID*), the sender-initiated rendezvous protocol (*SEND_RNDV*), and the receiver-initiated rendezvous protocol (*RECV_RNDV*). These protocols are then integrated with the eager protocol (*EAGER*) to form a complete point-to-point communication system. Hence, the full point-to-point communication system consists of four protocols and defines two message size thresholds, *EAGER_THRESHOLD* and *HYBRID_THRESHOLD*. Table 6.1 summarizes the protocols to be used in different situations.
Table 6.1: Protocols and the situations when they are used

<table>
<thead>
<tr>
<th>Message size</th>
<th>early party</th>
<th>protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$msize \leq \text{EAGER.THRESHOLD}$</td>
<td>sender</td>
<td>EAGER</td>
</tr>
<tr>
<td>(small message)</td>
<td>receiver</td>
<td>EAGER</td>
</tr>
<tr>
<td></td>
<td>both</td>
<td>EAGER</td>
</tr>
<tr>
<td>$\text{EAGER.THRESHOLD} &lt; msize$</td>
<td>sender</td>
<td>HYBRID</td>
</tr>
<tr>
<td>$msize \leq \text{HYBRID.THRESHOLD}$</td>
<td>receiver</td>
<td>RECV.RNDV</td>
</tr>
<tr>
<td>(medium message)</td>
<td>both</td>
<td>HYBRID</td>
</tr>
<tr>
<td>$\text{HYBRID.} \text{EAGER.THRESHOLD} &lt; msize$</td>
<td>sender</td>
<td>SEND.RNDV</td>
</tr>
<tr>
<td>(large message)</td>
<td>receiver</td>
<td>RECV.RNDV</td>
</tr>
<tr>
<td></td>
<td>both</td>
<td>RECV.RNDV</td>
</tr>
</tbody>
</table>

For small messages ($msize \leq \text{EAGER.THRESHOLD}$), the eager protocol is near optimal for all situations shown in Figure 6.1. For medium sized messages ($\text{EAGER.THRESHOLD} < msize \leq \text{HYBRID.THRESHOLD}$), two protocols are used. When the sender arrives early, the hybrid protocol shown in Figure 6.2a is used. In this protocol, the sender first copies the message data to a local buffer. After that, it checks if the receiver has arrived (RECEIVER READY message). If the receiver has arrived, the receiver-initiated protocol, shown in Figure 6.3a is used. Otherwise, the sender issues a SENDER READY message, which contains the address of the local buffer and other related information to facilitate the RDMA read from the receiver, to the receiver. The task in the sender side is completed; and the sender can exit the operation. When the receiver gets the SENDER READY message, it performs an RDMA read to obtain the data from the sender buffer. Note that the sender side buffer must be released at some point. However, since this is a library buffer that the application will not access, the information for the receiver to notify the sender that the buffer can be released can be piggybacked in a later control message.

In comparison to EAGER, HYBRID does not require buffering at the receiving side (does not have the $O(N)$ memory requirement) and removes the memory copy at the receiving side. These features enable HYBRID to efficiently transfer medium-sized messages that are larger than those handled by EAGER. Thus HYBRID pushes
the benefits of EAGER to medium-sized messages without additional resource requirements or unnecessary synchronization.

The hybrid protocol is also used when both parties arrive at similar times. The scenario is shown in Figure 6.2b. In this case, both SENDER_READY and RECEIVER_READY messages will be sent (and pass each other). Once the sender sends the SENDER_READY message, it completes the operation. The RECEIVER_READY will be ignored and dropped by the sender later. After the receiver sends RECEIVER_READY, it monitors both its data buffer and incoming control messages. If it sees a matching SENDER_READY, it performs the RDMA read operation to obtain the data.

When the receiver arrives early, it sends a RECEIVER_READY message to the sender, which carries the receiving buffer information. When the sender arrives, it can directly deposit the message into the receiver user space. Note that the RECEIVER_READY message is not in the critical path of the communication when the receiver arrives early. Hence, only the absolutely needed message transfer operation is in the critical path in this case: the protocol can deliver near optimal performance. The receiver-initiated protocol is depicted in Figure 6.3a.

For large messages ($HYBRID_THRESHOLD < msize$), the two cases are also separated. When the receiver arrives early, the receiver-initiated protocol shown in Figure 6.3a, is used. When the sender arrives early, a sender-initiated protocol, either the previously discussed RDMA write-based scheme in Figure 2.5a or the RDMA read-based scheme in Figure 2.5b, can be used. As discussed earlier, there is no clean-cut winner between these two protocols as either one can provide a better performance in different situations. the prototype system supports both protocols; either protocol can be pre-selected to handle the large messages when the sender arrives early. When both parties arrive at similar times, the receiver-initiated protocol is used: the SENDER_READY packet is ignored when the matching RECEIVER_READY packet has been sent. This scenario is depicted in Figure 6.3b. Notice that in this case, SENDER_READY is not in the critical path of the communication operation.
Figure 6.2: The hybrid protocol

(a) hybrid protocol (sender early)  (b) hybrid protocol (both)

Figure 6.3: The receiver-initiated rendezvous protocol
Table 6.2: Advantages of the integrated protocols over the traditional rendezvous protocol

<table>
<thead>
<tr>
<th>Early party</th>
<th>Medium sized message</th>
<th>Large sized message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>unnecessary synchronization removed, improved communication progress</td>
<td>NA</td>
</tr>
<tr>
<td>Receiver</td>
<td>simpler protocol, improved communication progress</td>
<td>simpler protocol, improved communication progress</td>
</tr>
<tr>
<td>Both</td>
<td>unnecessary synchronization removed; improved communication progress</td>
<td>number of protocol events in the critical path reduced</td>
</tr>
</tbody>
</table>

For a typical point-to-point communication where the receiver is receiving from one sender, the receiver-initiated protocol can be used for all cases with the sender never sending out the SENDER_READY message. I have discussed the two cases when the receiver arrives early and when both arrive at similar times. In the case when sender arrives early, it does not need to send the SENDER_READY message. Instead, it just needs to respond to the RECEIVER_READY message, and the performance is equivalent to the sender-initiated RDMA write-based protocol. However, the SENDER_READY message from the sender is essential for supporting MPI_ANY_SOURCE.

If one can integrate all these protocols into one communication system, the system will provide better communication performance than existing schemes that are based on the sender-initiated eager and rendezvous protocols. The advantages of the proposed integrated scheme are summarized in Table 6.2. For medium sized messages when the sender arrives early or both arrive at similar times, HYBRID behaves like EAGER and enjoys the advantages of EAGER over the rendezvous protocol. For medium sized messages and large messages when the receiver arrives early, the integrated system will use the receiver-initiated protocol that, in comparison to sender-initiated protocols, removes unnecessary synchronizations, reduces the number of rounds of control messages, and provides better communication progress. For large messages when both arrive at similar times, SENDER_READY is removed from the communication critical path, which results in a more efficient protocol.
6.2 Integrating the four protocols

I will now describe how the four protocols can be integrated without introducing excessive control overheads. Figure 6.4 gives the high level description of a system that simultaneously supports the four protocols. In the description, it is assumed that the receiver receives from one sender. The support of MPI\_ANY\_SOURCE will be discussed later. It is also assumed the buffers for eager messages and control messages are organized using the persistent buffer association technique [24] so that the sender always knows where to send an eager or control message and the receiver always knows where to poll the incoming eager and control messages. The sender-initiated protocol described in Figure 6.4 is the RDMA write-based rendezvous protocol. The RDMA read-based rendezvous protocol can also be used to replace the RDMA write-based protocol with minor modifications.

Figure 5.3 presents the communication system by describing the tasks in four critical points during the communication. Isend represents the time when the send operation starts (e.g. MPI\_Isend is called); Isend\_Wait represents the time when the send operation needs to be completed (e.g. MPI\_Wait is called on the MPI\_Isend); Irecv represents the time when the receive operation starts (e.g. MPI\_Irecv is called); Irecv\_Wait represents the time when the receive operation needs to be finished (e.g. MPI\_Wait is called on the MPI\_Irecv). In the description, some details about the protocol are omitted. For example, the SENDER\_READY message should contain the buffer address, message size, and other information so that the receiver can perform an RDMA read operation. The RECEIVER\_READY message should contain buffer address, and other information so that the sender can perform an RDMA write operation. It is also assumed that the preparations required for RDMA operations (e.g. memory registration) are done before the messages are sent out.

The communication system in Figure 5.3 seems to be somewhat straight-forward: the sender and the receiver perform protocol tasks based on the cases previously discussed. However, there are some subtleties in supporting the four protocols simultaneously. The subtleties include dealing with the mismatched protocols perceived
Isend:
(1) If \((\text{msize} \leq \text{EAGER\_THRESHOLD})\) /* small msg, EAGER*/
(2) Issue the command to send the eager message; return
(3) Else if \((\text{msize} \leq \text{HYBRID\_THRESHOLD})\) /* medium msg */
(4) Check outstanding remote messages (drop older messages);
(5) If (find a matching \text{RECEIVER\_READY} message) /* RECV\_RNDV */
(6) Issue RDMA write to remote address, send FIN; return;
(7) Else
(8) Copy message to buffer; check remote message (drop older msgs);
(9) If (find a matching \text{RECEIVER\_READY}) goto (6) /* RECV\_RNDV */
(10) Else {send \text{SENDER\_READY}; Done = 1; return;}
(11) Else /* large message */
(12) Check outstanding remote messages (drop older messages);
(13) If (find a matching \text{RECEIVER\_READY}) goto (6) /* RECV\_RNDV */
(14) Else {send \text{SENDER\_READY}; return;} /* HYBRID */
(15) Else /* large message */
(16) Check outstanding remote messages (drop older messages);
(17) If (find a matching \text{RECEIVER\_READY}) goto (6) /* RECV\_RNDV */
(18) Else {send \text{SENDER\_READY}; return;} /* starting SEND\_RNDV */

Isend\_Wait:
(1) If (Done == 1) return; /* HYBRID is done */
(2) If (EAGER message is outstanding for the communication)
(3) Wait for the completion of the eager message; return;
(4) Else if (RDMA write is outstanding for the communication) /* finishing RECV\_RNDV */
(5) Wait for the completion of the RDMA write message; return;
(6) Else /* finishing SEND\_RNDV */
(7) Wait for \text{RECEIVER\_READY}
(8) Issue RDMA write to remote address, send FIN;
(9) Wait for the RDMA write to complete; return;

Irecv:
(1) state = 0;
(2) If \((\text{msize} \leq \text{EAGER\_THRESHOLD})\) {return;}
(3) Check outstanding remote control message (drop older messages)
(4) If (find a matching \text{SENDER\_READY})
(5) If (message size \leq \text{HYBRID\_THRESHOLD}) /* HYBRID */
(6) Issue RDMA read to get the data; state = 1;return;
(7) Else /* SEND\_RNDV */
(8) Send \text{RECEIVER\_READY}; return;
(9) Send \text{RECEIVER\_READY}; return;

Irecv\_Wait:
(1) If (state == 1) {wait for the RDMA read to complete; return;}
(2) Poll control messages and eager messages
(3) If (receiving a matching eager message)
(4) Copy eager message to the user buffer; return;
(5) If (receiving a matching \text{SENDER\_READY} message)
(6) If (message size is less than \text{HYBRID\_THRESHOLD})
(7) Issue RDMA read to get the data;
(8) wait for the read to complete; return
(9) Else drop the message /* useless message */
(10) If (receiving a matching FIN message) return;
(11) Goto (2); /* loop until one of the above happens */

Figure 6.4: High level description of the integrated scheme
by the sender and the receiver, and matching control messages. I will now discuss these subtleties.

### 6.2.1 Dealing with mismatched protocols perceived by the sender and the receiver

MPI specification allows the message size posted by the sender to be different from that posted by the receiver. The only requirement is that the actual data size (posted by the sender) must be smaller than the size posted by the receiver. Hence, only the sender can decide the actual protocol. In the proposed scheme, when the receiver posts a size larger than the EAGER_THRESHOLD, if the receiver does not receive SENDER_READY from the sender, it will send the RECEIVER_READY message, and then anticipate all possibilities by monitoring both the control messages and the eager messages (Line (2) in Irecv_Wait). It can easily be verified that all possible combinations of mismatched protocols (e.g. receiver RECV_RNDV and sender EAGER) will not cause problems in my system except that one extra useless RECEIVER_READY message may be sent. Note that even with no protocol mismatch, one extra useless control message may still be sent in the integrated scheme.

### 6.2.2 Matching control messages

To support the four protocols in one framework, the control messages for each matching send and receive must be matched. This is critical since the proposed scheme allows great flexibility for the sender and the receiver to generate (sometimes useless) control messages. Again, in the following discussion, it is assumed that MPI_ANY_SOURCE and MPI_ANY_TAG are not used and each send/receiver operation is provided with a destination/source node and a tag. MPI_ANY_SOURCE and MPI_ANY_TAG will be discussed in the next subsection.

The deterministic MPI matching semantics are discussed in Section 2.2.1. To match control messages in the integrated scheme, each process keeps a counter to track the number of communications encountered during execution that can be associated
with a distinct tuple of \(<\text{communication type, communicator, tag, remote rank}>\). Communication type is either send or receive and remote rank represents either (1) the MPI rank of the receiver for a send or (2) the MPI rank of the sender for a receive. When a communication is encountered that necessitates a new tuple counter, such a counter is created and initialized to 1 (denoting communication at hand). Future communications associated with the same tuple increment the counter. In this way, communications can be enumerated in a fashion consistent with MPI matching semantics on both the sender and receiver. The header of each control message carries the counter to indicate its sequence. The receiver of a control message can compare the counter in the header with the counter that it maintains to decide whether the control message matches a given communication, is an old control message, or is for a future communication.

Consider an example where a receiver arrives early and posts two receives R1 and R2 with the same communicator, tag, and source where R1 is for a small message (using EAGER) and R2 is for a large message (using RECV_RNDV). R2 triggers a RECEIVERREADY message to be sent with a counter 2 (the second receive on the tuple). Now, when sender arrives with the matching first send S1 of the tag, it will find a RECEIVERREADY message with a matching tag. However, since the local counter is 1 (to send the first message), the sender will decide that the RECEIVERREADY is for a future communication, and will perform the communication using EAGER without matching the RECEIVERREADY message. After this communication, the sender counter for the tuple is incremented to 2 (the first message for this tuple has been sent). When the second send on the tuple is posted, the RECEIVERREADY will be matched correctly.

Assuming that a constant number of tags and communicators are used in MPI programs, maintaining the counters introduces \(O(n)\) memory overheads per process where \(n\) represents number of MPI processes. Since the size of each counter is only a few bytes and \(O(n)\) Infiniband connections are allocated, the counter scheme is as scalable as Infiniband itself.
6.2.3 Dealing with MPI\textunderscore ANY\textunderscore SOURCE and MPI\textunderscore ANY\textunderscore TAG

The receiver-initiated scheme cannot support MPI\textunderscore ANY\textunderscore SOURCE since the process does not know which sender is going to send a message. Since my system supports both the sender-initiated scheme and the receiver-initiated scheme and since only the receiver can post a receive operation with MPI\textunderscore ANY\textunderscore SOURCE, it can support MPI\textunderscore ANY\textunderscore SOURCE by disabling the receiver-initiated messages for receiving operations with MPI\textunderscore ANY\textunderscore SOURCE. By having the receiver react to the sender with the sender-initiated scheme, the counters for the number of messages for each tuple to the sender can also be updated once the sender identity is resolved. MPI\textunderscore ANY\textunderscore TAG poses a similar problem for updating the counters, and can be handled in a similar fashion. Basically, when MPI\textunderscore ANY\textunderscore SOURCE and MPI\textunderscore ANY\textunderscore TAG are used, the receiver-initiated mechanism is suppressed and a sender-initiated protocol is used until the wildcard is resolved.

6.2.4 Eliminating the FIN messages in common cases

In the receiver-initiated protocol, the last FIN message can be eliminated when the following two conditions are met: (1) the message size posted by the receiver matches the actual data size, and (2) receiving the last byte of a message indicates the arrival of the whole message. The first condition is common in MPI programs. The second condition is true in many contemporary systems including most InfiniBand clusters [24]. The receiver can put the message size that it anticipates and the value of the last byte in its buffer in the header of the RECEIVER\_READY message. The sender can then check (1) whether the receiver anticipated size matches the actual data size and (2) whether the last byte value in the receiving end is not equal to the last byte value in its data to be sent. If both tests are true, the sender can just RDMA write the data without sending the FIN: the receiver will monitor the last byte in its data buffer (in addition to the control messages and eager messages) and detect the completion of the operation when the last byte value changes.
### 6.3 Performance study

I have implemented MPI point-to-point communication routines over InfiniBand using the Verbs API [18] based on the proposed technique. Five MPI point-to-point routines, `MPI_Isend`, `MPI_Irecv`, `MPI_Send`, `MPI_Recv`, and `MPI_Wait`, and other routines including `MPI_Init()`, `MPI_Finalize()`, `MPI_Comm_rank()`, and `MPI_Comm_size()`, are supported by the implementation. My library can co-exist with MVAPICH [43]. When running an MPI application, the five MPI routine calls are modified in the application to invoke my routines, all other MPI routines that are not supported fall back to MVAPICH. The evaluation is done on an InfiniBand cluster, Draco, that has 16 compute nodes with a total of 128 cores. Each node is a Dell Poweredge 1950 with two 2.33Ghz Quad-core Xeon E5345’s (8 cores per node) and 8GB memory. All nodes run Linux with the 2.6.9-42.ELsm kernel. The compute nodes are connected by a 20Gbps InfiniBand DDR switch (CISCO SFS 7000D).

The performance of the integrated scheme is compared with MVAPICH2-1.2.rc1. My system supports either sender-initiated RDMA write-based protocol or sender-initiated RDMA read-based protocol (for large messages when the sender arrives early). MVAPICH2-1.2.rc1 also supports both the RDMA write-based rendezvous protocol and the RDMA read-based rendezvous protocol. Unfortunately, the MVAPICH RDMA read-based rendezvous protocol is not stable on the draco experimental cluster: it failed to run most micro-benchmarks and all application benchmarks. Hence, I will focus on comparing my scheme with the default RDMA write-based rendezvous protocol in MVAPICH2-1.2.rc1. To obtain a fair comparison, the proposed system also uses the RDMA write-based protocol when the sender arrives early. Note that there is no clear winner between these two sender-initiated protocols. Hence, improving over either protocol demonstrates the effectiveness of the proposed integrated scheme. The default EAGER_THRESHOLD in MVAPICH is used, which is 12KB. In my library, the EAGER_THRESHOLD is also set to 12KB and the HYBRID_THRESHOLD is set to 40KB. These two threshold values are the optimal cut-off points for the pingpong benchmark on the draco system.
6.3.1 Pingpong benchmark results

The pingpong benchmark tests the overall protocol efficiency when there is no computation-communication overlap opportunities. Figure 6.5 shows the results of the pingpong program. Each result in the figure is the average of five runs. The integrated scheme consistently out-performs MVAPICH for this program. For medium sized messages (16KB and 32KB), the differences are more significant. For example, for the 16KB message, per round trip time using my scheme is 55 micro-seconds versus 67 micro-seconds with MVAPICH. This is due to the use of the hybrid protocol, which is more efficient for this size. For larger message sizes (64KB, 128KB, 256KB), the performance of my scheme is also better. For all of the sizes, the integrated scheme is about 10 micro-seconds faster per iteration. Two factors contribute to this performance gain. First, using the integrated scheme, the SENDER_READY message is taken out of the critical path in the communication when the two processes arrive at a communication at similar times, and is eliminated when the receiver arrives early. Using monitoring counters in my system, I found that about 95% of the time, both parties arrive to the communications at similar times in this benchmark. Second, my scheme does not have the FIN message for this benchmark (eliminated by the technique discussed in Section 4.2.4). These results demonstrate that even without communication and computation overlap opportunities, the integrated approach introduces less communication overheads and achieves higher performance than the
traditional rendezvous protocol.

### 6.3.2 Progress benchmark results

I use the same progress micro-benchmark discussed in the last chapter, shown in Figure 5.8, to evaluate the performance of different protocols with different protocol invocation scenarios. Again, the time for 1000 iterations of the loop is measured and the notation $C(\text{comp}X)$ is used for the time for $\text{comp}X$ computations, $T(\text{msize})$ for the time to transfer a message of $\text{msize}$ bytes, and $\text{copy}(\text{msize})$ for the time to copy a message of $\text{msize}$ bytes. In the experiment, $C(X)+C(Y) \approx C(X+Y)$, $C(1) \approx 18\mu s$ and $T(100KB) \approx 90\mu s$.

Figure 6.6 shows the results for medium sized messages when the sender arrives early. Figure 6.6a show the results for the configurations $(0,0,60,X,0,0) = (10,20,30,-40,50,0,0)$ and 30KB message size. In this configuration, the sender has a total of 60 units of computation while the receiver has a total of $X$ units. As can be seen from the figure, the hybrid protocol completely eliminates the dependence from the sender to the receiver and as a result, the sender side computation can be completely overlapped with the communication and the receiver side computation, which results in a total time of roughly $C(60)$ for all different Xs. Using the traditional rendezvous
protocol, the sender cannot complete the communication until the receiver arrives at the operation. Hence, the sender side computation and receiver side computation are sequentialized in this configuration, resulting in a total per iteration time of roughly $C(60) + C(X) + T(msize)$, as shown in Figure 6.6a. Note that the penalty for unnecessary synchronizations depends on how the program is written and is independent of the time for transferring the data. Hence, the benefits for removing unnecessary synchronizations can potentially be very large. The hybrid protocol can also provide better communication progress at the receiver side. This is shown in Figure 6.6b with the configuration $(0, 20, 0, 5, 5, 20)$. In this case, the receiver has a total 30 units of computation. When the hybrid protocol is used, the communication is overlapped with the 5 units of computation after the $MPI.Irecv$ is called and the total per iteration time is roughly $C(30)$. Using the traditional rendezvous protocol, the data is moved at $MPI.Wait$ at the sender side, hence the total per iteration time is $C(20) + C(20) + T(msize) \approx C(40) + T(msize)$. These results are captured in Figure 6.6b. By eliminating unnecessary synchronizations, HYBRID can significantly improve the communication progress both at the sender and the receiver in comparison to the traditional rendezvous protocol.

Figure 6.7 shows the progress benchmark results for large messages with receiver arriving early (so that RECV_RNDV is used in my scheme). Figure 6.7a
contains the results for configuration \((20, 20, 20, 0, 0, 0)\). This case tests the sender side computation-communication overlap capability. As can be seen from the figure, MVAPICH is not able to overlap computation with communication in this case. The total per iteration time is \(C(60) + T(\text{msize})\), which increases linearly with the message size. On the other hand, the integrated scheme uses RECV_RNDV for this communication and is able to overlap \text{comp2} with the communication. The per iteration time is thus \(C(40) + \max\{C(20), T(\text{msize})\}\), which increases only when \(\text{msize} > 400\,\text{KB}\) and \(T(\text{msize}) > C(20)\). Figure 6.7b contains the results for configuration \((27, 0, 0, 20, 20, 20)\). This case tests the receiver side computation-communication overlap capability. As can be seen from the figure, the traditional rendezvous protocol is not able to overlap computation with communication in this case (per iteration time is still \(C(60) + T(\text{msize})\)) while the integrated scheme is able to overlap the communication and computation and has much lower total times for this program \((C(47) + \max\{C(13), T(\text{msize})\})\).

The results in this subsection demonstrate that for many common situations the integrated approach provides much better performance than the traditional rendezvous protocol. In particular, by using customized protocols, the proposed scheme reduces the unnecessary synchronization and improves communication progress at both the sender and receiver sides, which can result in significant performance gains.

### 6.3.3 Application benchmark results

In this section, I present results from experiments using NAS applications benchmarks (NPB2.4) [44]. NPB2.4 has 6 point-to-point communication benchmarks: BT, CG, EP, LU, MG, and SP. EP does not have many communications. The communications in LU and MG are mainly performed by MPI_Irecv with MPI_ANY_SOURCE and my system uses the same protocol as MVAPICH. Hence, I will only discuss results for BT, CG, and SP.

Table 6.3 shows the total running times, total communication times, and the communication improvement percentages using the integrated scheme over MVAPICH for
Table 6.3: Performance of application benchmark (all times are in seconds)

<table>
<thead>
<tr>
<th>Bench.</th>
<th>Class</th>
<th>N</th>
<th>MVAPICH</th>
<th>Integrated</th>
<th>Comm. improv. percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>total</td>
<td>comm.</td>
<td>total</td>
</tr>
<tr>
<td>BT</td>
<td>A</td>
<td>9</td>
<td>32.95</td>
<td>1.41</td>
<td>32.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>18.40</td>
<td>1.29</td>
<td>17.38</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>9</td>
<td>137.21</td>
<td>2.99</td>
<td>136.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>79.12</td>
<td>4.14</td>
<td>76.84</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>9</td>
<td>565.39</td>
<td>7.49</td>
<td>563.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>321.76</td>
<td>10.11</td>
<td>315.51</td>
</tr>
<tr>
<td>CG</td>
<td>A</td>
<td>8</td>
<td>0.71</td>
<td>0.14</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0.43</td>
<td>0.13</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>8</td>
<td>25.43</td>
<td>2.05</td>
<td>25.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>14.34</td>
<td>1.89</td>
<td>14.30</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>8</td>
<td>64.92</td>
<td>3.73</td>
<td>64.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>35.66</td>
<td>3.41</td>
<td>35.46</td>
</tr>
<tr>
<td>SP</td>
<td>A</td>
<td>9</td>
<td>16.48</td>
<td>1.08</td>
<td>16.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>9.46</td>
<td>0.96</td>
<td>9.14</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>9</td>
<td>76.38</td>
<td>2.58</td>
<td>75.18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>41.51</td>
<td>2.83</td>
<td>39.95</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>9</td>
<td>325.25</td>
<td>6.99</td>
<td>322.35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>180.57</td>
<td>6.53</td>
<td>177.22</td>
</tr>
</tbody>
</table>
CLASS A, B, C of the three programs running on 8/9 and 16 processes. The communication time includes all Send, Isend, Recv, Irecv, and Wait times, which account for the majority of all communication times in these benchmarks. Since my system does not have optimizations for intra-node communication within a SMP node, there cannot be a fair comparison of the protocols for intra-node communications. Hence, the experiments are run with one process on each node so that all communications go through the InfiniBand interface. It must be noted that this experiment is designed to demonstrate that the proposed combined protocol is more efficient than the traditional rendezvous protocol, not to indicate that my system is better than MVAPICH.

As can be seen from the table, the integrated scheme obtains very significant improvement over MVAPICH for SP and BT (all classes) in terms of the communication time. This is because there are lots of communication and computation overlapping opportunities in these two benchmarks. The integrated scheme is much better in exploiting such opportunities, which was also demonstrated in the progressive benchmark results. Although communication does not account for a large percentage of the total application time in these benchmarks, the improvement in communication times transfers into improvement of the overall benchmarks time, as shown in the table. The CG benchmark does not have much communication-computation overlap opportunities. The integrated scheme still noticeably improves the communication performance in this program.

6.4 Chapter Summary

The techniques discussed in this chapter provide protocol customization without modification to the MPI application or the way the code is executed. I show how the eager, hybrid, sender-initiated and receiver-initiated RDMA protocols can be integrated into a novel dynamic protocol selection scheme that selects protocols that are optimized to the run-time characteristics of a particular communication. I further elaborate how such a scheme can support MPI wildcards and remove unnecessary
protocol messages in the common case. Compared to traditional MPI libraries with a static protocol, dynamic protocol customization is able to reduce unnecessary synchronizations, decrease the number of control messages in the critical communication path, and improve the communication progress.

To evaluate its potential performance advantages, I implement the proposed scheme for Infiniband clusters. The ping-pong benchmark results demonstrate that dynamic selection slightly reduces message latency compared to the sender-initiated protocol even when there is not opportunity for communication-computation overlap. The progress benchmark results shows that for some communication arrival patterns, dynamic protocol selection can significantly outperform MVAPICH, fully overlapping communication and computation. Finally, the application benchmark results reveal that my scheme can improve communication performance by as much as 156% compared to MVAPICH.
CHAPTER 7

CONCLUSION

I propose and evaluate techniques that support the optimization of MPI point-to-point communication for RDMA-enabled clusters by leveraging the flexibility and high performance of RDMA. My research in designing efficient small message channels demonstrates that small message channel memory usage and scalability can be significantly improved by removing the need for persistent buffer association and by adopting a novel shared small message channel scheme. The experimental results confirm that while coalescing fragmented memory resources and reducing the number of channels, my shared scheme does not sacrifice the performance benefits of RDMA. My research in developing near-optimal rendezvous protocols map new and existing rendezvous protocols to protocol invocation scenarios for which they are collectively near-optimal for all scenarios. The experimental results show that when the near-optimal protocols are used properly, communication performance can be significantly improved. This provides a foundation for the future profile-driven, compiler-assisted protocol customization research. My dynamic protocol selection research aims to apply protocol customization at run-time, without the need to use profile data or use compiler techniques. I develop a scheme that uses process arrival information and message size to inform dynamic protocol selection and that integrates several protocols into a single system while still supporting MPI semantics. The performance results show that the dynamic protocol selection is effective in reducing synchronization, improving communication progress and increasing the overlap of communication and computation.
BIBLIOGRAPHY


http://www.mpi-forum.org/docs/


[34] Message Passing Interface Forum. MPI_Isend Documentation.  


[38] Message Passing Interface Forum. MPI_Scan Documentation. 


[40] Message Passing Interface Forum. MPI_Send Documentation. 


[43] MVAPICH: MPI over InfiniBand and iWARP, 
http://mvapich.cse.ohio-state.edu.


BIOGRAPHICAL SKETCH

Matthew Small was born on April 2, 1984 in Syracuse, NY and spent his childhood and formative years in the Central New York region. He attended the University of South Florida and graduated with honors in the spring of 2007, earning his Bachelors degree in Computer Engineering. The following fall, he came to Florida State University to pursue his PhD in computer science as a Graduate Assistants in Areas of National Need (GAANN) fellow. Throughout his doctoral studies, Matthew’s research focus primarily dealt with improving communication performance for scientific application through the optimization of communication libraries.