High Performance MPICH2 One-Sided Communication Implementation over InfiniBand

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Weihang Jiang, B.E.

* * * * *

The Ohio State University

2004

ABSTRACT

Due to its high performance and open standard, InfiniBand is gaining popularity in the area of high performance computing. In this area, MPI is the *de facto* standard for writing parallel applications. One-sided communication (Remote Memory Access) is one of the major extensions indicated in MPI-2. MPICH2, as a successor of MPICH, features a completely new design which aims to support MPI-2. In this thesis, we present a study of implementing MPICH2 one-sided communication for InfiniBand clusters.

In many existing MPI-2 one-sided communication implementations (including original MPICH2), one-sided communication are built on top of send/receive functions, and thread based designs are used for implementing passive synchronization. Although this approach can achieve good portability, it suffers from the overhead and constraints brought by two-sided communication and threaded implementation.

To address these problems, we propose a new design for MPI-2 one-sided communication by exploiting features of the InfiniBand Architecture. In our design, MPI-2 one-sided communication functions such as MPI_Put, MPI_Get and MPI_Accumulate are directly mapped to InfiniBand Remote Direct Memory Access (RDMA) operations. For synchronization functions, we eliminate the involvement of thread by using the algorithms that make advantage of InfiniBand remote atomic operations.

Our design has been implemented based on MPICH2 over InfiniBand. We present detailed design issues for both our new approaches and original approaches. A set of micro-benchmarks are performed to characterize different aspects of performance. The experimental results have shown that, comparing with send/receive based design, RDMA based design can achieve lower overhead and higher communication performance. Moreover, the RDMA based design allows for better overlap between computation and communication and achieves better scalability with multiple number of origin processes. For synchronization functions, our experimental results show that the atomic operation based designs can achieve less synchronization overhead, better concurrency, and consume less computing resource comparing with the thread based designs.

I dedicate this work to my father Quhu Jiang and my mother Ying Xu

ACKNOWLEDGMENTS

I would like to express my sincere appreciation and gratitude to my advisor Dr. Dhabaleswar K. Panda for his constant guidance and motivation. His valuable support has gone a long way in the completion of this work.

I am grateful to Dr. Mario Lauria for consenting to serve on my Master's examination committee.

For generously sharing their thoughts, ideas and comments, I would like to acknowledge Jiuxing Liu, Hyun-Wook Jin, Jiesheng Wu, Pavan Balaji and Amith Mamidala.

I am thankful to Dr. William Gropp, Dr. Darius Buntinas , Dr. Rajeev Thakur and other MPICH2 group members from Argonne National Laboratory for their involvement in this work.

My heartfelt thanks to my family and friends for their support, love and affection.

VITA

December, 1979	Born - Fuzhou, China
1998 - 2002	B.E. Computer Science and Engineering,Zhejiang University
Sept 2003 - Present	Graduate Research Associate, Ohio State University.

PUBLICATIONS

Research Publications

"Efficient Implementation of MPI-2 Passive One-Sided Communication over InfiniBand Clusters", Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, Darius Buntinas, Rajeev Thakur and William Gropp, *Euro PVM/MPI* , 2004.

"High Performance MPI-2 One-Sided Communication over InfiniBand", Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda , William Gropp and Rajeev Thakur, *IEEE/ACM International Symposium on Cluster Computing and the Grid* , 2004.

"Design and Implementation of MPICH2 over InfiniBand with RDMA Support", J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp and B. Toonen, *International Parallel and Distributed Processing Symposium* , 2004.

"Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics", Jiuxing Liu, Balasubramanian Chandrasekaran, Jiesheng Wu, Weihang Jiang, Sushmitha Kini, Weikuan Yu, Darius Buntinas, Pete Wyckoff and D. K. Panda, *Supercomputing Conference* , 2003.

FIELDS OF STUDY

Major Field: Computer and Information Science

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

One of the important characteristics of advances in computing technology is that they enable new capabilities to explore science. The development of Network of Workstations (NOW) is one example. Network interconnects that offer low latency and high bandwidth, complementing commodity workstations with boosted processing power, provide a cost-effective solution with high computational abilities. Significant cost advantages, combined with better scalability, have made NOW a more attractive solution for high performance scientific computing than traditional Supercomputers. In 22nd edition of "Top500" list [46] of world's fastest supercomputers, the list of NOW systems in TOP10 has grown impressively to 7 systems.

This chapter starts with a brief overview of modern interconnects and communication protocols for NOW systems. After that, this chapter describes several programming models for writing parallel application on NOW. And then we focus on Message Passing Interface (MPI)-2, which supports one-sided communication model (Remote Memory Access). Then we come up with the problem statement, which is about implementing high performance one-sided communication over InfiniBand. We finish this chapter with an outline of this thesis.

1.1 Interconnect Technologies and Communication Protocol

Network interconnects significantly affect the scalability of a NOW system to achieve expected performance. Vendors have invested a lot of resources in developing interconnect products, which include InfiniBand Architecture (IBA) [22, 5], Myrinet [9], Gigabit Ethernet [15, 28], 10 Gigabit Ethernet [1], Quadrics [2, 39, 40] and GigaNet CLAN [13, 14]. Most of these interconnects can deliver low latency (about several μs), and high bandwidth (more than 1 Gbps). Besides that, some of these network interconnects provide totally new features at hardware level, and open new avenues to design NOW systems.

In traditional UNIX communication protocols, messages go through the kernel and involve several copies. As the high-speed interconnects reach Gbps level, the bottleneck in communication has been shifted from limited bandwidth of network fabrics to communication protocol software stack. Consequently, various protocols like U-Net [45], VMMC [8], AM [47], EMP [42, 43], FM [38], GM [35] etc. with specified network interfaces have been proposed to bypass the kernel. The U-Net is the first user-level communication protocol proposed by T. Eicken et al. which allows user-level access to network interface device to avoid extra copies. The VMMC is another user-level protocol which separates data transfer from control transfer (i.e. without receiving node CPU support). Inspired by these research projects, Compaq, Intel and Microsoft developed the preliminary Virtual Interface Architecture (VIA) specification [12, 6]. InfiniBand Architecture, which mentioned above, is an implementation and extension of VIA.

1.2 Parallel Programming Models

Basing on the architecture of the target systems, parallel programs can be written using various programming models. For NOW systems with distributed memory, Message Passing and Shared Memory are the standard parallel programming models, and Remote Memory Access is an new model, which can complement Message Passing model.

Message Passing Interface (MPI) [31, 44, 20] is the representative of Message Passing model. An MPI program creates multiple processes, with each process maintains its local data. The processes are identified by a unique name, and processes cooperate by sending and receiving messages to and from each others. Message Passing model can offer scalability and good performance, however explicit communications make it too complex to code for some applications.

Distributed Shared Memory (DSM) [23] provides a logically Shared Memory Model over physically distributed memory systems. The processes in a DSM program share a common address space. From programmers' point of view, the advantage of this model is that there is no need to specify communication explicitly. Therefore, DSM can simplify program development. However, the performance of a DSM program is hard to be deterministic, for that the locality of data is difficult to understand and manage.

Combining advantages of both Message Passing and Shared Memory Models, Remote Memory Access model simplifies program development while remains the concept of local memory. MPI-2 (An extension to MPI) [19, 32] and A Portable Remote Memory Copy Interface (ARMCI) [36] are two standards supporting RMA model.

1.3 Message Passing Interface-2

MPI is the *de facto* standard for writing parallel applications. As an extension to MPI, MPI-2 introduces several new features. One-sided communication, dynamic process management and parallel I/O are the major new functionalities of MPI-2.

In the traditional two-sided communication model, both parties must perform matching communication operations (e.g., a *send* and a *receive*). In one-sided communication model, a matching operation is not required from the remote party. All parameters for the operation, such as source and destination buffers, are provided by the initiator of the operation. One-sided communication model can support more flexible communication patterns and improve performance in certain applications.

In MPI-2, dynamic process model allows processes to be created and terminated after an MPI application has started. A new mechanism is added to establish communication between the newly created processes and the existing MPI processes. It also includes a mechanism to establish communication between separately started MPI processes.

Different from POSIX interface, MPI-2 parallel I/O interface is a high-level interface that supports partitioning of file data among processes and a collective transfer interface between process memories and files. Further efficiencies can be achieved by supporting asynchronous I/O, strided accesses, and control over physical file layout on storage devices.

1.4 Problem Statement

MPI-2 one-sided communication model decouples synchronization and data transfer. Therefore, explicit synchronization is required to guarantee the completion of data transfer functions. MPI-2 one-sided communication supports three data transfer functions and two synchronization modes: active and passive. In active mode, the target is actively involved in synchronization, whereas in passive mode, the target node is not explicitly involved in synchronization.

To implement one-sided communication with active synchronization, the key is to implement efficient data transfer functions. One common way to implement data transfer functions in MPI-2 one-sided communication is to use existing MPI two-sided communication operations such as *send* and *receive*. This approach has been used in several popular MPI implementations, including the current MPICH2 [3] and SUN MPI [11]. Although this approach can get good portability, it suffers from the protocol overhead and progress dependence brought by two-sided communication.

To implement one-sided communication with passive synchronization, the most important issue is to implement efficient passive synchronization functions. Most implementations of MPI-2 passive one-sided communication are implemented based on the thread based designs [11, 33, 34, 48, 30, 4]. The thread based design can achieve good portability, because it is based on two-sided message passing, which is supported by all platforms. However, thread based designs can not handle concurrent communications well. Another problem of thread based designs is that since the assisting thread keeps running at a target process, CPU cycles are consumed by the thread even if there is no passive one-sided communication.

Recently, InfiniBand has entered the high performance computing market. InfiniBand architecture supports Remote Direct Memory Access (RDMA) and remote atomic operations. RDMA operations enable direct access to the address space of a remote process and their semantics match quite well with MPI-2 one-sided communication. The remote atomic operations, which are also supported by InfiniBand at hardware level, allow us to implement the synchronization algorithms designed for shared memory environment to distributed private memory environment efficiently. Therefore, in this work, we aim to provide answers to the following two questions:

1. *Can we optimize the MPI-2 one-sided data transfer functions by using schemes that leverage the RDMA operations in InfiniBand architecture?*

2. *Instead of using thread based designs, can we improve performance of passive one-sided communication by using algorithms that take advantage of the remote atomic operations in InfiniBand architecture?*

The rest of this thesis is organized as follows. In Chapter 2 we provide an overview of the InfiniBand Architecture and the various features it provides. We also take a look at MPI-2 one-sided communication and the MPICH2 implementation of the MPI-2 standard for InfiniBand clusters. Chapter 3 focuses on the designs and performance evaluation of MPI-2 one-sided data transfer functions and active synchronization functions. In Chapter 4, after analyzing the design issues in implementing passive synchronization functions, we show different designs for passive synchronization and their performance. In chapter 5 we draw our conclusions and discuss future work.

CHAPTER 2

BACKGROUND

In this chapter we provide an overview of InfiniBand Architecture and the set of features that can be utilized for the efficient implementation of one-sided communication. After that, we give a brief overview of MPI-2 one-sided communication model. Then, we describe an MPI-2 implementation: MPICH2 [3]. As a background of later work, we also explain how we implement MPICH2 over InfiniBand. Finally, we briefly describe the related work.

2.1 InfiniBand Architecture

The InfiniBand Architecture (IBA) defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). Channel Adapters usually have programmable DMA engines with protection features. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs sit on processing nodes. Their semantic interface to consumers is specified in the form of InfiniBand Verbs. TCAs connect I/O nodes to the fabric. Their interface to consumers are usually implementation specific and thus not defined in the InfiniBand specification.

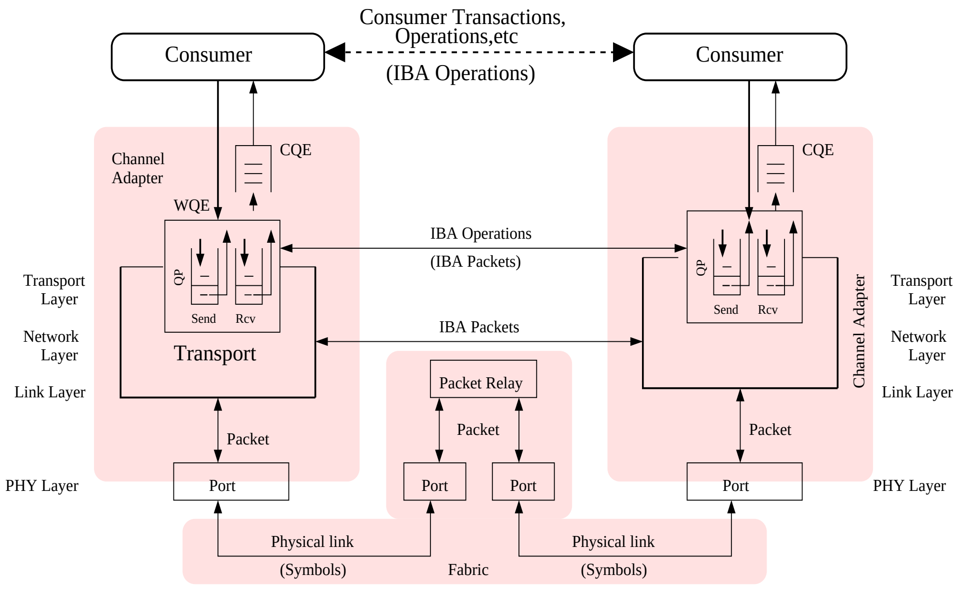

Figure 2.1: InfiniBand Communication Architecture

Figure 2.1 shows the InfiniBand communication architecture. The communication stack consists of different layers. The interface presented by Channel adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A Queue Pair in InfiniBand Architecture consists of two queues: a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication operations are described in Work Queue Requests (WQR), or descriptors, and submitted to the work queue. Once submitted, a Work Queue Request becomes a Work Queue Element (WQE). WQEs are executed by Channel Adapters. The completion of work queue elements is reported through Completion Queues (CQs).

Once a work queue element is finished, a completion queue entry is placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished. InfiniBand also supports different classes of transport services.

For communication, InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. To receive a message, the programmer posts a receive descriptor which describes where the message should be put at the receiver side. At the sender side, the programmer initiates the send operation by posting a send descriptor. The send descriptor describes where the source data is but does not specify the destination address at the receiver side. When the message arrives at the receiver side, the hardware uses the information in the receive descriptor to put data in the destination buffer. Multiple send and receive descriptors can be posted and they are consumed in FIFO order. The completion of descriptors are reported through CQs.

2.1.1 RDMA in InfiniBand Architecture

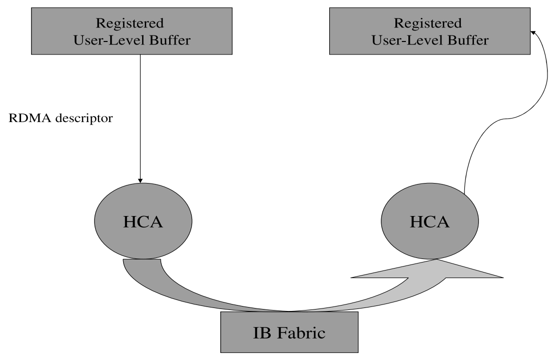

Figure 2.2: InfiniBand Communication Architecture

InfiniBand also supports memory semantic operations, called Remote Direct Memory Access (RDMA). RDMA operations are one-sided and do not incur software overhead at the other side. As shown in Figure 2.2, the sender (initiator) starts RDMA by posting RDMA descriptors. A descriptor contains both the local data source addresses (multiple data segments can be specified at the source) and the remote data destination address. At the sender side, the completion of an RDMA operation can be reported through CQs. The operation is transparent to the software layer at the receiver side. Since RDMA operations enable a process to access the address space of another process directly, they have raised some security concerns. In InfiniBand architecture, a key based mechanism is used. A memory buffer must first be registered before they can be used for communication. Among other things, the registration generates a remote key. This remote key must be presented when the sender initiates an RDMA operation to access the buffer.

Compared with send/receive operations, RDMA operations have several advantages. First, RDMA operations themselves are generally faster than send/receive operations because they are simpler at the hardware level. Second, they do not involve managing and posting descriptors at the receiver side, which incur additional overheads and reduce the communication performance. However, if we use RDMA, the destination address and protection key must be known beforehand.

RDMA operations include RDMA write and RDMA read:

1. RDMA Write is a memory semantic operation that allows a process to write a virtually contiguous buffer on a remote node. This is a one-sided operation that does not incur a software overhead at the remote host.

2. RDMA Read is a memory semantic operation that allows a process to read a virtually contiguous buffer on a remote node and write to a local memory buffer.

2.1.2 Atomic Operations in InfiniBand Architecture

Another emerging feature of InfiniBand Architecture is remote atomic operation. Two 64-bit atomic operations are supported: Compare-and-Swap and Fetch-and-Add. The Compare-and-Swap operation reads a 64-bit content from the memory in an remote process, compares the content with the parameter (compare_value) of this atomic operation, and puts the value of another parameter (swap_value) into the remote memory if the two compared values are the same. The Fetch-and-Add operation reads data from the remote memory, makes an addition operation between the data and the parameter (add_value) of this atomic operation, and updates the result to the remote memory. Both Compare-and-Swap and Fetch-and-Add operations bring back the old value of the variable in the remote memory. As the name "atomic operation" suggests, Compare-and-Swap and Fetch-and-Add operations are handled atomically, and more importantly, they are processed by the processor on the HCAs, without the intervention of CPU at the remote side.

2.2 MPI-2 One-Sided Communication

MPI-2 one-sided communication model, which is also called Remote Memory Access model, allows only one side to provide all the information for the communication. Therefore, it allows programmers to use more flexible communication pattern by avoiding matching send functions with receive functions.

We describe some terms commonly used in one-sided communication here. Then we give a detailed explanation for it in both active and passive modes.

Origin Process is the process that performs the one-sided communication functions.

Target Process is the process in which the memory is accessed by an origin process through one-sided communication functions.

Window is the memory area in target process to which an origin process can access through the one-sided communication.

Three one-sided communication functions are defined in MPI-2 specification. They are MPI_Put, MPI_Get and MPI_Accumulate. As showed in Figure 2.3:

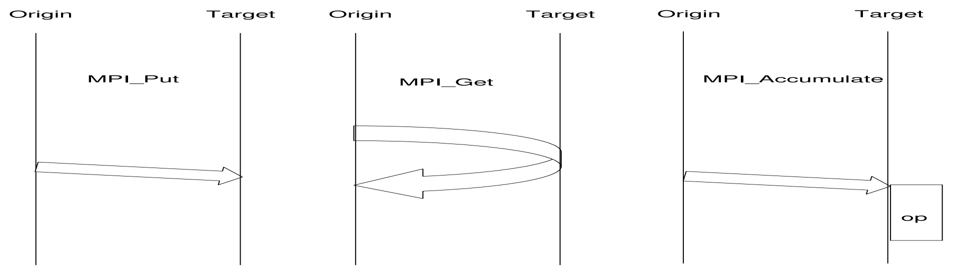

Figure 2.3: MPI-2 One-Sided Communication functions

MPI_Put is the function that transfers the data from an origin process to a window at a target process.

MPI_Get is the function that transfers the data from a window at a target process to an origin process.

MPI_Accumulate is the function that combines the data movement to a target process with a reduce operation.

In MPI-2, explicit synchronization functions must be used to guarantee the completion of the communication functions. Synchronization functions are classified as *active* and *passive*. The active synchronization involves both sides of communication while passive synchronization only involves the origin side.

2.2.1 One-Sided Communication with Active Synchronization

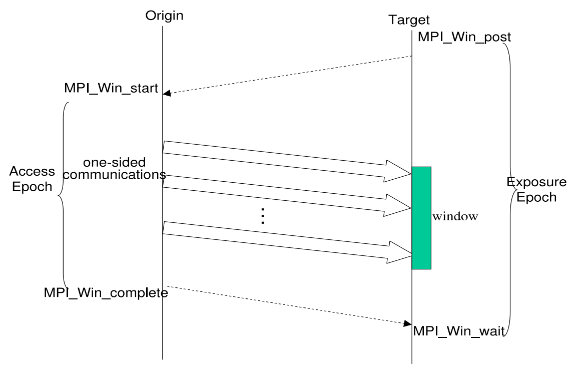

Figure 2.4: MPI-2 One-Sided Communication with Active Synchronization

In Figure 2.4, we use an example to explain MPI-2 one-sided communication with active synchronization. We can see that synchronization is achieved through four MPI functions: MPI_Win_start, MPI_Win_complete, MPI_Win_post and MPI_Win_wait. MPI_Win_post and MPI_Win_wait specify an *exposure epoch* in which other processes can access a memory window in this process. MPI_Win_start and MPI_Win_complete specify an *access epoch* in which the current process can use one-side communication operations such as MPI_Put to access memory in the target process. Multiple operations can be issued in the access epoch to amortize the overhead of synchronization. The completion of these operations are not guaranteed until the MPI_Win_complete returns. The active synchronization can also be achieved through MPI_Win_fence.

2.2.2 One-Sided Communication with Passive Synchronization

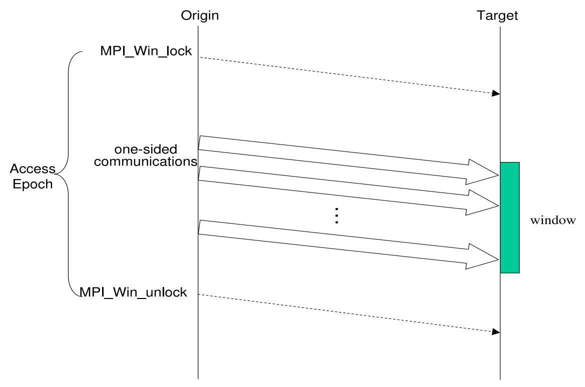

Figure 2.5: MPI-2 One-Sided Communication with Passive Synchronization

Figure 2.5 shows an example of passive one-sided communication. The origin process uses MPI_WIN_Lock to acquire the access control on the window in the target process. Then, multiple one-sided communication functions can be issued from the origin process to access the window protected by the lock, which the origin process just acquires. Finally, the origin process calls MPI_WIN_Unlock, to guarantee the completion of data transfer and release the lock.

As we can see, locks are used in passive one-sided communication model to protect accesses to target window. Only one window can be specified by these functions, and one or multiple one-sided communication data transfer functions can be issued to the target window between MPI_Win_lock and MPI_Win_unlock. Shared and exclusive locks are provided by the two functions MPI_Win_lock and MPI_Win_unlock. Accesses protected by an exclusive lock can not be concurrent at the window with other accesses to the same widow, while accesses protected by a shared lock can be concurrent at the window with other accesses protected by a shared lock to the same window , but it can not be concurrent with other accesses protected by an exclusive lock.

Close to a shared memory model, in passive one-sided communication model, shared data can be accessed by all processes. Different origin processes can communicate with each other by accessing the same memory inside target window. Target process can be different from these origin processes, in which case it is not explicitly involved in the communication or the synchronization.

2.3 MPI-2 Implementation – MPICH2

MPICH [18] is developed at Argonne National Laboratory. It is one of the most popular MPI implementations. The original MPICH provides supports for MPI-1 standard. As a successor of MPICH, MPICH2 supports not only MPI-1 standard, but also functionalities such as dynamic process management, one-sided communication and MPI I/O, which are specified in MPI-2 standard. However, MPICH2 is not merely MPICH with MPI-2 extensions. It is based on a completely new design, aiming to provide more performance, flexibility and portability than the original MPICH.

2.3.1 MPICH2 Structure and RDMA Channel interface

One of the objectives in MPICH2 design is portability. To facilitate porting MPICH2 from one platform to another, communication-related functionalities are encapsulated in an interface call ADI3 (the third generation of Abstract Device Interface). ADI3 contains a large number of functions and it is not a trivial task to port it from one interconnect to another. To simplify the task, a smaller, channel-based interface call CH3 (the third generation of the Channel interface) is introduced [41].

CH3 has many of the performance advantages of ADI3 interface. However, it only contains about a dozen functions and re-targeting it to a different communication platform requires considerably less effort.

To take advantage of architectures with globally shared memory or RDMA capabilities and further reduce the porting overhead, MPICH2 introduces another interface called RDMA Channel interface below the CH3 interface.

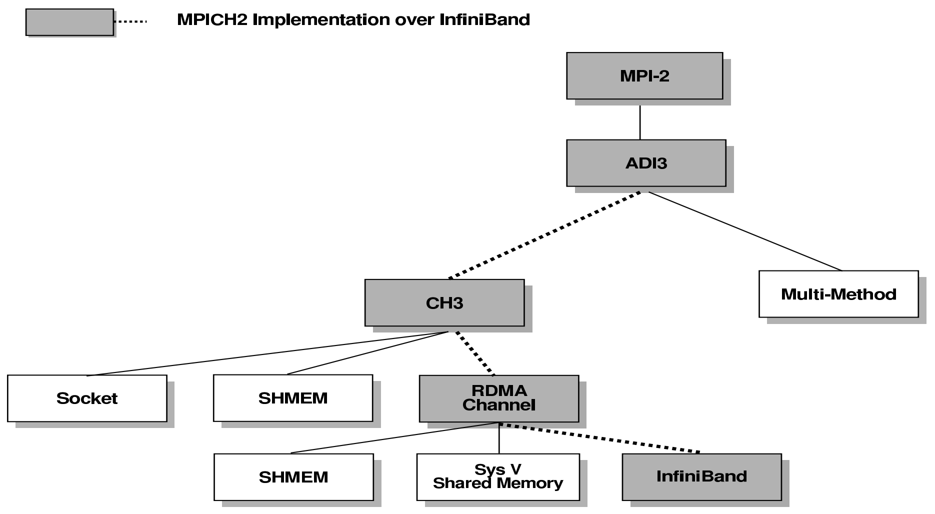

Figure 2.6: MPICH2 Implementation Structure

The hierarchical structure of MPICH2 , as shown in Figure 2.6, gives much flexibility to implementors. The three interfaces (ADI3, CH3, and RDMA Channel Interface) provide different tradeoffs between communication performance and ease of porting. Currently, there exist implementations based on all the three interfaces.

RDMA Channel interface is designed specifically for architectures with globally shared memory or RDMA capabilities. It contains five functions, among which only two are central to communication. (Other functions deal with process management, initialization and finalization.) These two functions are called *put* and *get*.

Both *put* and *get* functions accept a connection structure and a list of buffers as parameters. They return to the caller the number of bytes that have been successfully put or got. If the bytes completed is less than the total length of buffers, the caller shall retry the same get or put operation later.

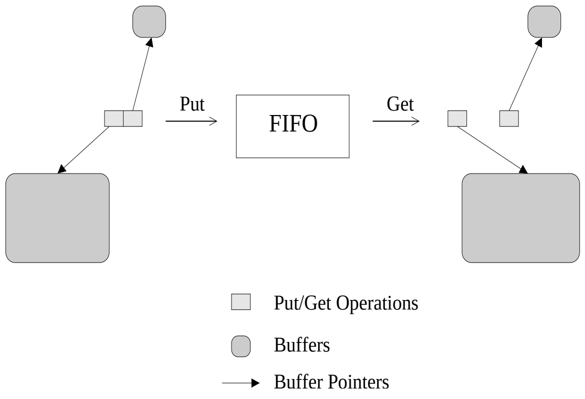

Figure 2.7: Put and Get Operations

Figure 2.7 illustrates the semantics of *put* and *get*. Logically, a pipe is shared between the sender and the receiver. The *put* operation writes to the pipe and the *get* operation reads from it. The data in the pipe is consumed in FIFO order. Both operations are non-blocking in the sense that they return immediately with the number of bytes completed instead of waiting for the entire operation to finish.

Put and *get* operations can be implemented on architectures with globally shared memory in a straightforward manner. Figure 2.8 shows one example. In this implementation, a shared buffer (organized logically as a ring) is placed in shared memory, together with a head pointer and a tail pointer. The *put* operation writes the buffer

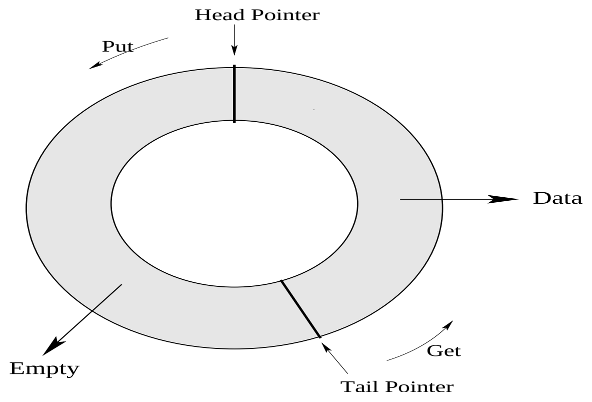

Figure 2.8: Put and Get Implementation with Globally Shared Memory

to the shared buffer and adjust the head pointer. The *get* operation involves reading from the shared buffer and adjusting the tail pointer. In the case of buffer overflow or underflow (detected by comparing head and tail pointers), the operations return immediately and the caller will retry them.

2.3.2 MPICH2 Implementation over InfiniBand

Because of the the hierarchical structure of MPICH2, we can port MPICH2 to InfiniBand cluster by implementing RDMA Channel Interface with the communication operations supported by InfiniBand.

In our previous work [27], we present a study of using RDMA operations to implement MPICH2 over InfiniBand. Our work takes advantage of the RDMA Channel interface provided by MPICH2.

As we mentioned above, the RDMA Channel interface provides a very small set of functions to encapsulate the underlying communication layer upon which the whole MPICH2 implementation is built. Consisting of only five functions, the RDMA Channel Interface is easy to implement for different communication architectures. However, the question arises whether this abstraction is powerful enough so that one can still achieve good performance.

Our study has shown that the RDMA Channel interface still provides the implementors with much flexibility. With optimizations such as piggybacking, pipelining and zero-copy, MPICH2 is able to deliver good performance to the application layer. For example, one of our designs achieves 7.6μs latency and 857MB/s peak bandwidth, which come quite close to the raw performance of InfiniBand.

2.4 Related Work

Besides MPI-2, there are other programming models that uses one-sided communication. Some of the examples are ARMCI [36], BSP [17] and GASNET [10]. These programming models use one-sided communication as the primary communication approach while in MPI, both one-sided and two-sided communication are supported.

There have been studies regarding implementing one-sided communication in MPI-2. Similar to the current MPICH2, work in MPI-SUN [11] describes an implementation based on MPI two-sided communication. MPI-2 one-sided communication has also been implemented by taking advantage of globally shared memory in some architectures such as NEC SX [24]. For distributed memory systems, some of the existing studies, such as Fujitsu MPI, MPICH-SCI, LAM/MPI over VIA, have exploited the ability of remotely accessing another process's address space provided by the interconnect to implement MPI-2 one-sided operations [4, 48, 7]. For passive synchronization functions, most implementations mentioned above are implemented based on thread based designs [11, 33, 34, 48, 30, 4]. Work in [24, 29] describe a non-threaded NEC MPI/SX implementation of MPI-2 passive one-sided communication, where the progress engine takes charge of performing synchronization. This design does not satisfy the requirements we have defined in Section 3. The latest version of LAM/MPI [26] supports a part of MPI-2 functions that do not include the passive one-sided communication yet.

In this thesis, our target architecture is InfiniBand, which provides very flexible RDMA as well as atomic operations. We focus on the performance improvement of using these operations compared with the send/receive based approach for data transfer and the thread based approach for synchronization.

Work in [16] provides a performance comparison of several existing MPI-2 implementations. They have used a ping-pong benchmark to evaluate one-sided communication. However, their results do not include the MPICH2 implementation. In our work, we focus on MPICH2 and introduce a suite of micro-benchmarks which provide a more comprehensive analysis of MPI-2 one-sided operations, including communication and synchronization performance, communication/computation overlap, dependency on remote process and scalability.

2.5 Summary

In this chapter we provided an overview of the features and architecture of InfiniBand based clusters. We also provided an overview of the MPI-2 One-Sided Communication model. We described the architecture of MPICH2, an implementation of MPI-2, and show how we implement it over InfiniBand. Finally, we briefly introduce the related work about one-sided communication and our contribution.

CHAPTER 3

MPI-2 ONE-SIDED COMMUNICATION WITH ACTIVE SYNCHRONIZATION

3.1 Motivation

One common way to implement MPI-2 one-sided communication is to use existing MPI two-sided communication operations such as MPI_Send and MPI_Recv. This approach has been used in several popular MPI implementations, including the current MPICH2 and SUN MPI [11]. Although this approach can improve portability, it has some potential problems:

- Protocol overhead: Two-sided communication operations incur many overheads such as memory copy, matching of send and receive operations and handshake in Rendezvous protocol. These overheads decrease communication performance for one-sided communication.

- Dependency on remote process: The communication progress of one-sided communication operations are dependent on the remote process in this approach. As a result, process skew may significantly degrade communication performance.

As we have introduced, InfiniBand architecture supports Remote Direct Memory Access (RDMA). RDMA operations enable direct access to the address space of a remote process and their semantics match quite well with MPI-2 one-sided communication. In our recent work [27], we have proposed a design which uses RDMA to implement MPI two-sided communication. However, since its MPI-2 one-sided communication operations are implemented on top of two-sided communication, this implementation still suffers from the problems mentioned above.

Therefore, we want to see if we can optimize the MPI-2 one-sided communication with active synchronization by using the design that leverages the RDMA operations in InfiniBand architecture instead of using the existing MPI point-to-point operations.

3.2 Send/Receive Based MPI-2 One-Sided Communication Design

As we have mentioned, MPI-2 one-sided communication can be implemented using MPI two-sided communication operations such as MPI_Send, MPI_Recv and their variations (MPI_Isend, MPI_Irecv, MPI_Wait, etc.). (In the following discussions, we use "send" and "receive" to refer to different forms of MPI_Send and MPI_Recv, respectively.) We call this *send/receive based* approach. The current MPICH2 implementation uses such an approach. In this section, we will discuss MPICH2 as an implementation example of one-sided communication. The discussion is based on MPICH2 over InfiniBand (MVAPICH2)[1] [27].

3.2.1 Communication Operations

For the MPI_Put operation, the origin process first sends information about this operation to the target. This information includes target address, data type information, etc. Then the data itself is also sent to the target. After receiving the operation

[1]The current MVAPICH2 implementation is based on MPICH2 version 0.96p1. MVAPICH2 is available from [37].

information, the target uses another receive operation for the data. In order to perform the MPI_Get operation, first the origin process sends a request to the target, which informs it the data location, data type and length. After receiving the request, the target process sends the requested data to the origin process. The origin finishes the operation by receiving the data to its local buffer. For MPI_Accumulate, the origin process uses a similar approach to send the data to the target process. Then the target receives the data and performs a local reduce operation.

The send/receive based approach has very good portability. Since it only depends on MPI two-sided communication, its implementation is completely platform independent. However, it also has many drawbacks. First, it suffers from high protocol overhead in MPI send/receive operations. For example, MPI send/receive operations use Rendezvous protocol for large messages. In order to achieve zero-copy, the current MPICH2 uses a handshake in the Rendezvous protocol to exchange buffer addresses. However, since in one-sided communication, target buffer address information is available at the origin process, this handshake is unnecessary and brings degradation of communication performance. In addition, MPI send/receive may involve other overheads such as send/receive matching and extra copies.

Since the target is actively involved in the send/receive based approach, the overhead at the target process increases. The target process may become a performance bottleneck because of this increased overhead.

The send/receive based approach also makes the origin process and the target process tightly coupled in communication. The communication of origin process depends heavily on the target to make progress. As a result, process skew between the target and the origin may significantly degrade communication performance.

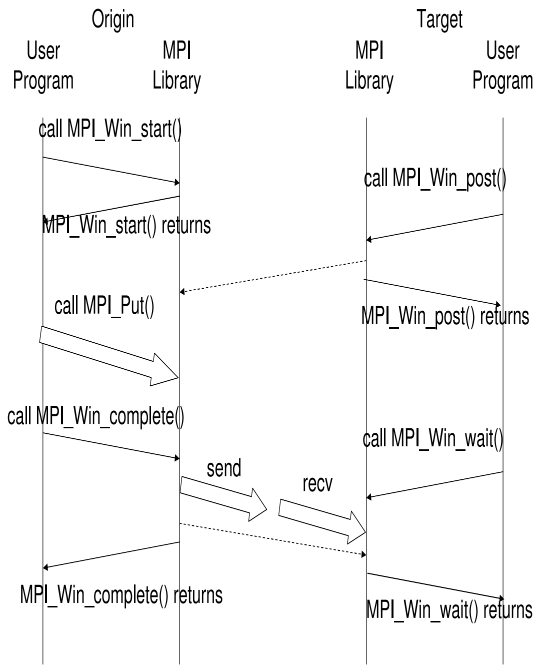

Figure 3.1: Send/Receive Based One-Sided Communication Implementation

3.2.2 Active Synchronization Operations

In MPI-2 one-sided communication, synchronization can be done using MPI_Win_start, MPI_Win_complete, MPI_Win_post and MPI_Win_wait. At the origin side, communication is guaranteed to finish only after MPI_Complete. Therefore, implementors have a lot of flexibility with respect to when to carry out the actual communication. In the send/receive based approach, communication involves both sides. Since the information about the communication is only available at the origin side, the target needs to be explicitly informed about this information. One way to address this problem in send/receive based approaches is to delay communication until MPI_Win_complete. Within this function, the origin sends information about all possible operations. In

MPI_Win_wait, the target receives this information and takes appropriate actions. An example of send/receive based implementation is shown in Figure 3.1.

Delayed communication used in send/receive based approach allows for certain optimizations such as combining of small messages to reduce per-message overhead. However, since the actual communication does not start until MPI_Complete, the communication cannot be overlapped with computation done in the access epoch. This may lead to degraded overall application performance.

In the current MPICH2 design, the actual communication starts when there are enough one-sided communication operations posted to cover the cost of synchronization. This design can potentially take advantage of the optimization opportunities in delayed communication and also allow for communication/computation overlap. However, since one-sided communication is built on top of send/receive, the actual overlap depends on how the underlying send/receive operations are implemented. In many MPI implementations, sending/receiving a large message goes through Rendezvous protocol, which needs host intervention for a handshake process before the data transfer. In these cases, good communication/computation overlap is difficult to achieve.

3.3 RDMA Based MPI-2 One-Sided Communication Design

As we have described, one-sided communication in MPICH2 is currently implemented based on send/receive operations. Therefore, it still suffers from the limitation of the two-sided communication design even though the MVAPICH2 [27]. In this section, we discuss how to utilize InfiniBand features such as RDMA operations to address these potential problems.

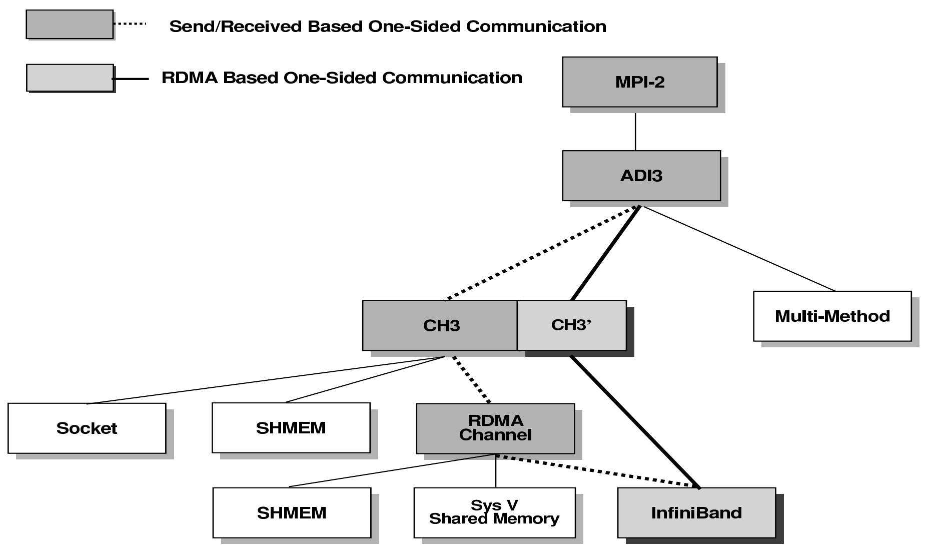

Figure 3.2: Design Architecture

MPICH2 has a flexible layered architecture in which implementations can be done at different levels. Our MVAPICH2 implementation over InfiniBand [27] [37] was done using the RDMA Channel Interface. However, this interface currently does not provide us with direct access to all the RDMA and atomic functions in InfiniBand. To address this issue, we use a customized interface *CH3 extension* to expose these functionalities to the upper layer and implement our design directly over this interface. The basic structures of our design and the original design are shown in Figure 3.2.

3.3.1 Communication Operations

We implement the MPI_Put operation with RDMA write. Through exchanging memory addresses at window creation time, we can keep record of all target memory addresses on all origin processes. When MPI_Put is called, an RDMA write operation is used, which directly transfers data from memory in the origin process to remote memory in the target process, without involving the target process. The MPI_Get operation is implemented with the RDMA read operation in InfiniBand. Based on InfiniBand RDMA and atomic operations, we have designed the accumulate operation as follows: The origin fetches the remote data from target using RDMA read, performs a reduce operation, and updates remote data by using RDMA write. Since there may be more than one origins, we use the Compare-and-Swap atomic operation to ensure mutual exclusive access.

By using RDMA, we can avoid protocol overhead of two-sided communication. For example, the handshake in Rendezvous protocol is avoided. Also, the matching between send and receive operations is no longer needed. So the overhead associated with unexpected/expected message queue maintenance, tag matching and flow control is eliminated.

More importantly, the dependency on remote process for communication progress is reduced. Unlike the send/receive based approach, using RDMA operations directly does not involve the remote process. Therefore, the communication can make progress even when the remote process is doing computation. As a result, our implementation suffers much less from process skew. Moreover, our design exploiting RDMA operations can make implementation of passive one-sided communication much easier because the target is not required to respond to one-sided communication operations.

3.3.2 Event Driven Accumulate Operations

One potential problem in RDMA based design is that an accumulate operation needs to fetch the remote operand and then write back the result, while in send/receive based design, an accumulate operation only transfers the local operand to remote

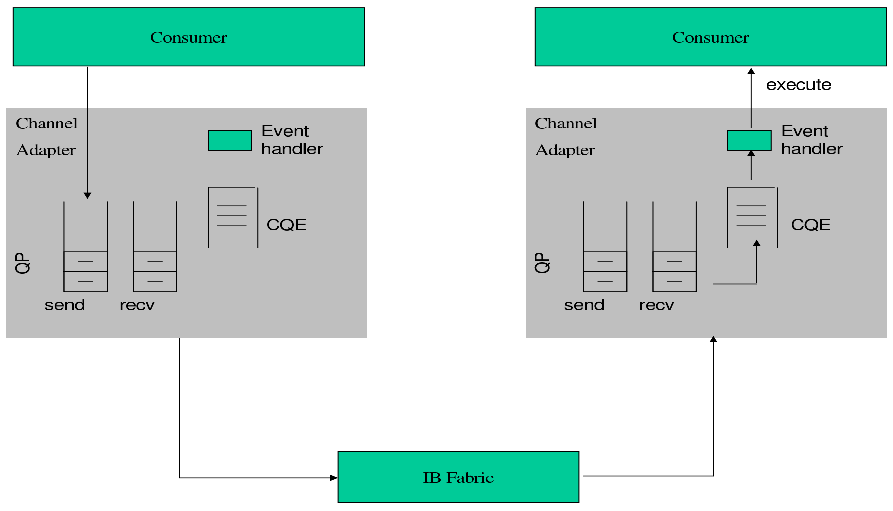

Figure 3.3: Event Driven Accumulate Operation

process. If the size of operand is large, this may cause significant overhead. Here we investigate an event driven based solution to see if it can optimize the performance of accumulate operation.

As we have mentioned in the background chapter, InfiniBand provides both channel and memory semantics. By using channel semantics, one process can generate a signal at a remote process. Also, InfiniBand HCA supports event handlers, which can be driven by signal.

Combining these features, we propose an event driven based design for accumulate operation. As shown in Figure 3.3, at the very beginning, a pre-defined event handler which can handle reduction functions is registered to HCA, and a receive descriptor is also posted. If a process wants to execute an accumulate operation on a remote process, it first posts a send descriptor matching with the pre-post receive descriptor at the remote party, to trigger the preregistered function. Then it sends the local operand to the remote party trough RDMA Write. Then the function at the remote side will execute reduce operation, and update the local memory. At the end, an acknowledgment message must be sent back to indicate the completion of accumulate function.

As we will see, Event Driven Accumulate Operation can achieve better performance when the size of the operand is very large. More details will be discussed later.

3.3.3 Active Synchronization Operations

In some send/receive based designs, actual communication is delayed until MPI _Win_complete is called. In our design, the one-sided communication will start as soon as the post operation is called. In our implementation, the origin process maintains a bit vector. Each bit represents the status of a target. A target uses RDMA write to change the corresponding bit. By checking the bits, the origin process can get synchronization information and start communications.

Targets can not leave MPI_Win_wait until communication has been finished. Therefore origin processes need to inform the targets after they have completed communication. For this purpose we also use RDMA write to achieve better performance. Before leaving the MPI_Win_wait operation, the targets check to make sure all origin processes have completed communication. An example of this RDMA based implementation is shown in Figure 3.4.

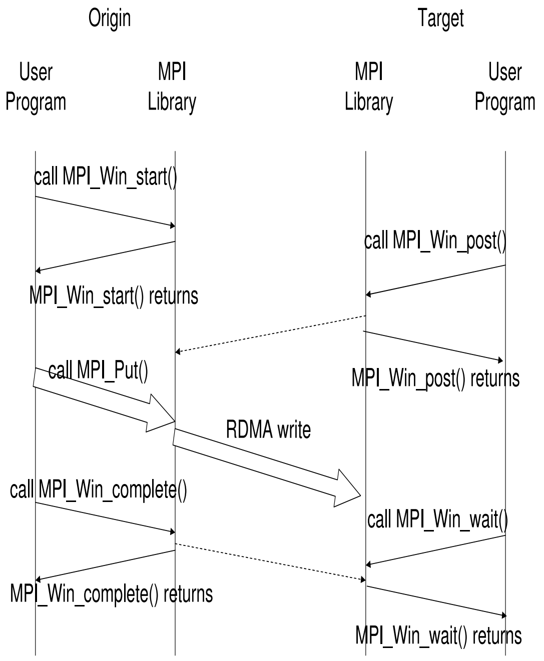

Figure 3.4: RDMA Based One-Sided Communication Implementation

3.3.4 Other Design Issues

By using RDMA, we potentially can achieve better performance. However, it also introduces several design challenges.

An important issue we should consider in exploiting RDMA operations is the memory registration. Before performing any RDMA operation, both source and destination data buffers need to be registered. The memory registration is an expensive operation. Therefore, it can degrade communication performance significantly if done in the critical path. All memory buffers for the one-sided communication on the target processes are declared when the window is created. Thus, we can avoid memory registration overheads by registering the memory buffers at the window creation time. For memory buffers at the origin side, pin-down cache [21] is used to avoid registration overhead for large messages. For small messages, we copy them to a pre-registered buffer to avoid registration cost.

Another important issue is to handle user-defined data type. The original approach requires data type processing at the target side. With RDMA operations, we can avoid this overhead by initiating multiple RDMA operations. Currently, our design only deals with simple data types. For complicated non-contiguous data types, we fall back on the original send/receive based implementation.

3.4 Performance Evaluation

In this section, we perform a set of micro-benchmarks to evaluate the performance of our RDMA based MPI-2 one-sided communication design and compare them with the original design in MPICH2. We have considered various aspects of MPI-2 one-sided communication such as synchronization overhead, data transfer performance, communication and computation overlap, dependency on remote process and scalability with multiple origin processes.

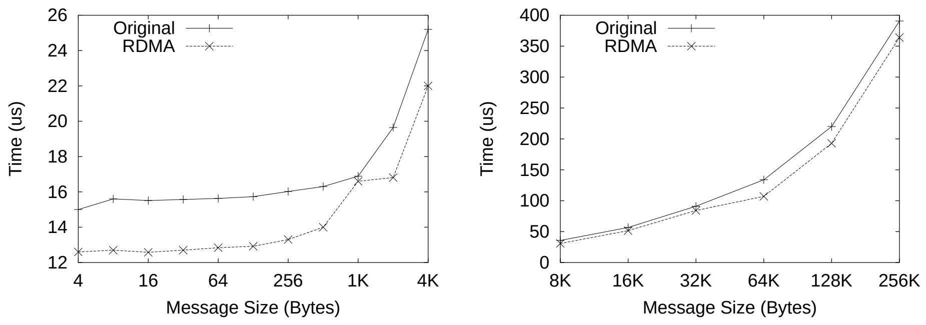

Figure 3.5: Ping-Pong Latency

We focus on active one-sided communication in the performance evaluation. Our tests use MPI_Win_start, MPI_Win_complete, MPI_Win_post and MPI_Win_wait functions for synchronization. However, most of the conclusions in this section are also applicable to programs using MPI_Win_fence.

3.4.1 Experimental Testbed

Our experimental testbed consists of a cluster system with 8 SuperMicro SUPER P4DL6 nodes. Each node has dual Intel Xeon 2.40 GHz processors with a 512K L2 cache and a 400 MHz front side bus. The machines are connected by Mellanox InfiniHost MT23108 DualPort 4X HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The HCA adapters work under the PCI-X 64-bit 133MHz interfaces. We used the Linux Red Hat 7.2 operating system with 2.4.7 kernel. The compilers we used were GNU GCC 2.96 and GNU FORTRAN 0.5.26.

3.4.2 Latency

For MPI two-sided communication, a ping-pong latency test is often used to characterize its performance. In this subsection, we use a similar test for MPI-2 one-side communication. The test consists of multiple iterations using two processes. Each iteration consists of two epochs. In the first epoch, the first process does an MPI_Put operation. In the second epoch, the second process does an MPI_Put operation. We then report the time taken for each epoch.

Figure 3.5 compares the ping-pong latency of our RDMA based design with the original MPICH2. We can see that the RDMA based approach can improve the latency. For small messages, it reduces latency from 15.6μs to 12.6μs (19% improvement). For large messages, since the handshake in Rendezvous protocol is avoided, it also gives better performance. The improvement is up to 17μs.

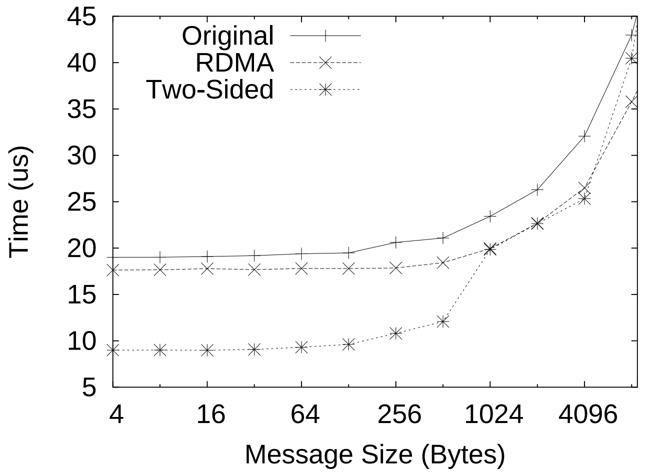

Figure 3.6: Bi-Directional Latency

A bi-directional latency test is often used to compare the performance of one-sided communication to two-sided communication. In this test, both sides send a message to the other side. In the one-sided version, the test is done using MPI_Put and MPI_Win_fence. In the two-sided version, the test is done using MPI_Isend, MPI_Irecv and MPI_Waitall. Figure 3.6 shows the performance results. We can observe that for very small messages, two-sided communication performs better because it does not use explicit synchronization. For one-sided communication, our RDMA based design always performs better than the original design. For messages larger than 4KB, it even outperforms two-sided communication.

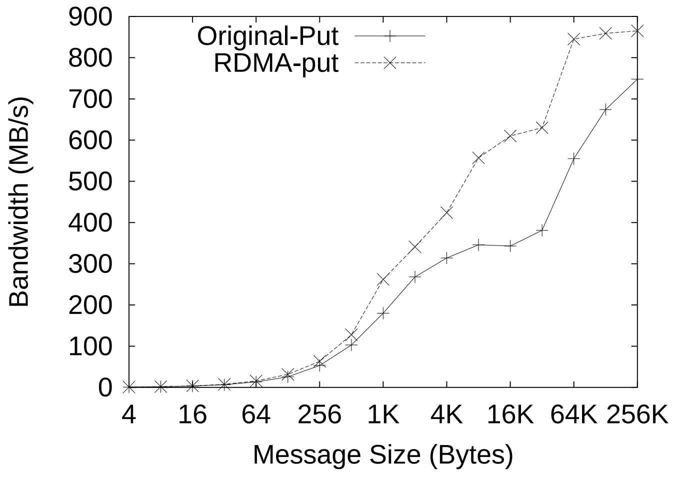

Figure 3.7: MPI_Put Bandwidth

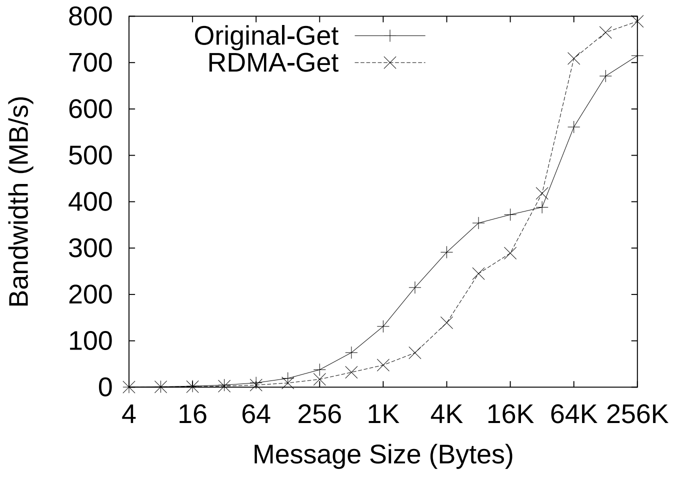

Figure 3.8: MPI_Get Bandwidth

3.4.3 Bandwidth

In applications using MPI-2 one sided operations, usually multiple communication operations are issued in each epoch to amortize the synchronization overhead. Our bandwidth test can be used to characterize performance in this scenario. This test consists of multiple epochs. In each epoch, *W* MPI_Put or MPI_Get operations are issued where *W* is a pre-defined burst size.

Figures 3.7 and 3.8 show the bandwidth results of MPI_Put and MPI_Get with a burst size(*W*) of 16. We can see that the RDMA based approach always performs better for MPI_Put. The improvement can be up to 77% for certain message size. For 256KB messages, it delivers a bandwidth of 865MB/s. (Note that unless stated otherwise, the unit MB in this paper is an abbreviation for 10^6 bytes.)

However, we also observe that the RDMA based approach does not perform as well as the original approach for MPI_Get with small messages. This is because RDMA read is used in our new design for MPI_Get while the original approach uses RDMA write. The bandwidth drop is due to the performance difference between InfiniBand RDMA read and RDMA write.

3.4.4 Synchronization Overhead

In MPI-2 one-sided communication, synchronization must be done explicitly to make sure data transfer has been finished. Therefore, the overhead of synchronization has great impact on communication performance. To characterize this overhead, we use a simple test which calls only MPI-2 synchronization functions (MPI_Win_start, MPI_Win_complete, MPI_Win_post and MPI_Win_wait) for multiple iterations.

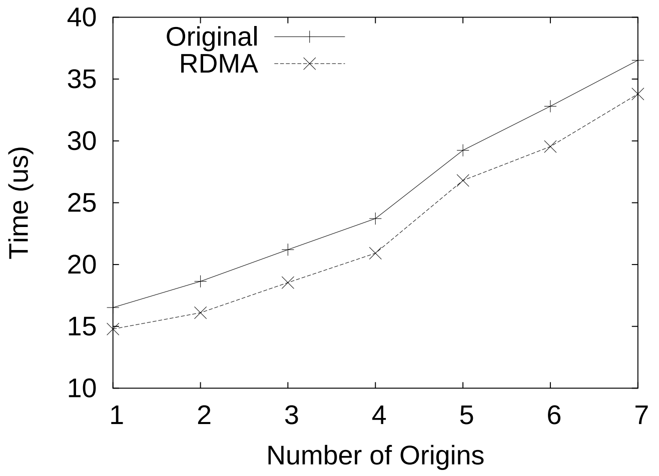

Figure 3.9: Synchronization Overhead

Figure 3.9 shows the time taken for each iteration for the original design and our RDMA based design. We can see that our new design slightly reduces synchronization time. When there is one origin, synchronization time is reduced from 16.52 microseconds to 14.78 microseconds (13% improvement). This is because we use InfiniBand level RDMA operations instead of calling MPI send and receive functions for synchronization. We have also done the test for one origin process with multiple target processes and the results are similar to Figure 3.9.

3.4.5 Communication/Computation Overlap

As we have mentioned, by using RDMA, we can possibly achieve better overlapping of communication and computation, which may lead to improved application performance. In this subsection, we have designed an overlap test to measure the ability to overlap communication and computation for different one-sided communication implementations.

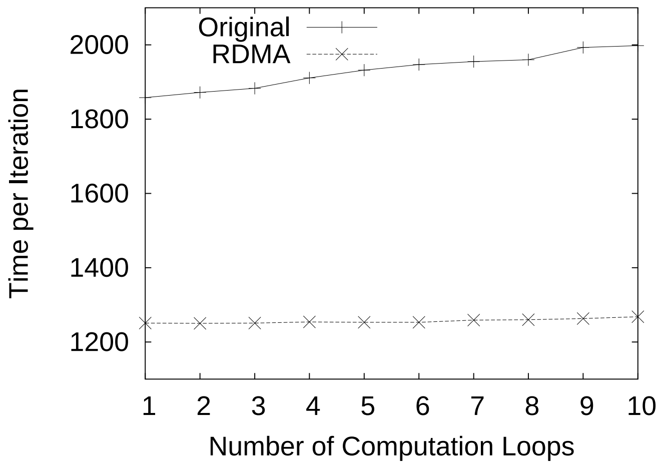

Figure 3.10: Overlap Test Results

The overlap test is very similar to the bandwidth test. The difference is that we have inserted a number of computation loops after each communication operation. Each computation loop increases a counter for 1,000 times. Figure 3.10 shows how the average time for one iteration of the test changes when we increase the number of computation loops for 64KB messages. We can see that the RDMA based design allows overlap of communication and computation and therefore its performance is not affected by increasing computation time. However, the original design shows lower performance when the computation increases.

3.4.6 Impact of Process Skew

As we have discussed, one of the advantages of using InfiniBand RDMA to implement MPI-2 one-sided communication is that the communication can make progress without depending on the target process. Therefore, skew between the origin and the target process will have less impact on the communication performance. Our process

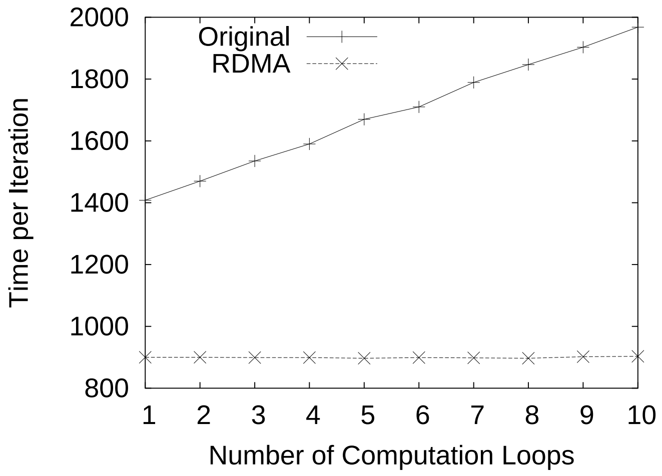

Figure 3.11: Process Skew Test Results

skew test is based on the bandwidth test. Process skew is emulated by adding different number of computation loops (with each loop increasing a counter for 10,000 times) between MPI_Win_post and MPI_Win_wait in the target process.

Figure 3.11 shows the performance results for 64KB messages. We can see that process skew does not affect the RDMA based approach at all. However, the performance of the original design drops significantly with the increase of process skew.

3.4.7 Performance with Multiple Origin Processes

Scalability is very important for MPI-2 designs. In MPI-2 one-sided communication, it is possible for multiple origin processes to communicate with a single target process. Figure 3.12 shows the aggregated bandwidth of all origin processes in this scenario. Here we use 64KB as message size and 16 as burst size (W). We should note that the aggregated bandwidth is limited by the PCI-X bus at the target node. We can observe that since the RDMA based design incurs very little overhead at the

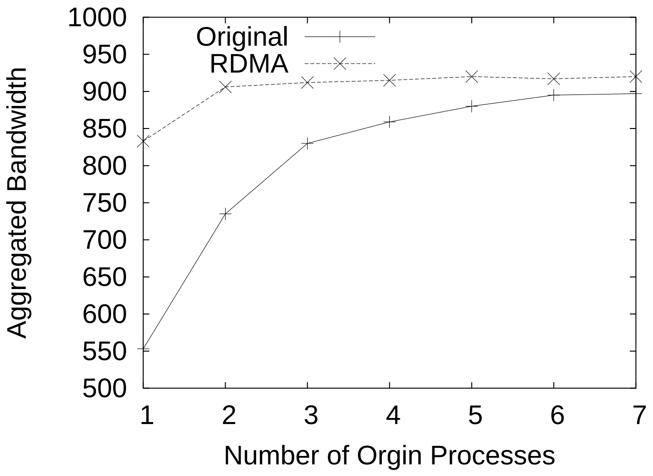

Figure 3.12: Aggregated Bandwidth with Multiple Origin Processes

target process, it reaches a peak bandwidth of over 920MB/s even with a small number of origin processes. The original design can only deliver a maximum bandwidth of 895MB/s.

3.4.8 Performance of Accumulate Operation

As we mentioned above, event driven accumulate operation can avoid moving the result of the reduce function. Figure 3.13 shows the time spent on one accumulate operation with different operand sizes. We can see that the event driven accumulate operation outperforms RDMA based accumulate operation when the number of integers in the operand is larger than 512 (2k Bytes). The reason for such a large crosspoint is that currently, it takes around 20 microseconds to trigger a preregistered function, and the acknowledgment message also takes some time.

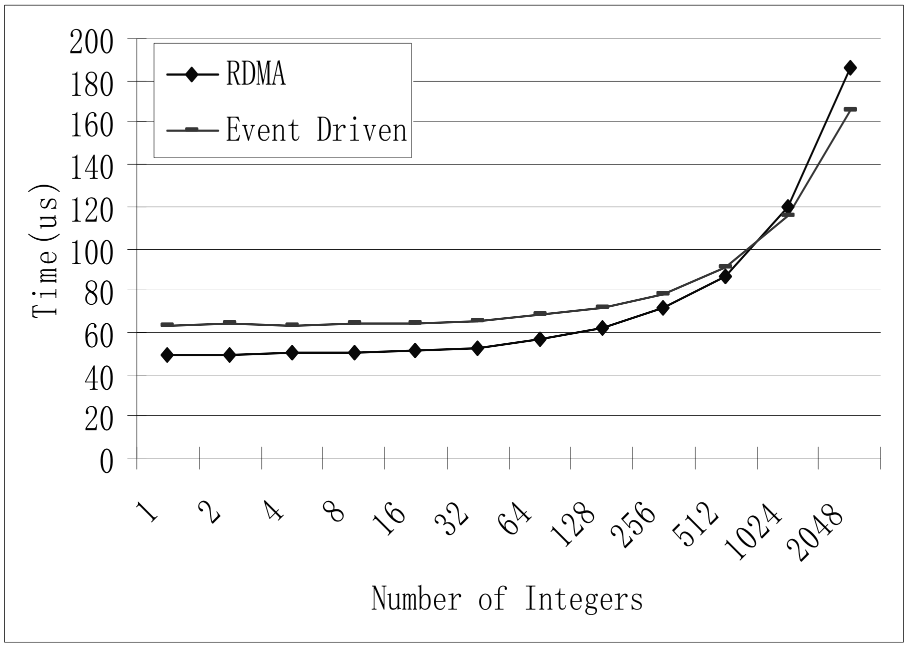

Figure 3.13: Performance of Accumulate Operation

3.5 Summary

In this chapter, we have proposed a design of MPI-2 one-sided communication with active synchronization over InfiniBand. This design eliminates the involvement of targets in one-sided communication completely by utilizing InfiniBand RDMA operations. Through performance evaluation, we have shown that our design can achieve lower overhead and higher communication performance. Moreover, experimental results have shown that the RDMA based approach allows for better overlap between computation and communication. It also achieves better scalability with multiple number of origin processes.

CHAPTER 4

MPI-2 ONE-SIDED COMMUNICATION WITH PASSIVE SYNCHRONIZATION

We have previously implemented MPI-2 active mode one-sided communication using DMA-based communication. In this chapter we extend this implementation to allow passive mode one-sided communication. The main challenge in extending active mode to passive mode is in designing an efficient synchronization mechanism, which allows one-sided operations to be performed at the target node independently of what the target process is doing.

We implement and evaluate four designs for passive mode synchronization. Since thread based design has been used in some MPI implementations, such as [11] and [34], to support MPI-2 one-sided communication, we do a complete study to examine thread based designs for passive synchronization functions. First, we use a dedicated thread, which uses a separate thread at the target process to handle incoming one-sided operations. We also evaluated a thread-based design using event driven blocking. In this design, we use the event handler and completion signal features of InfiniBand to allow the thread to block while idle to reduce its CPU utilization.

In the other two designs we use the remote atomic memory operations provided by InfiniBand. InfiniBand provides remote atomic Compare&Swap and Fetch&Add

operations. Many algorithms have been proposed for synchronization functions in shared memory machines. "Test and Set" may be the most widely used algorithm. Another algorithm is the MCS [25] algorithm proposed by John Mellor-Crummey and Michael Scott, which has been thought of as a classic scalable synchronization algorithm for shared memory multiprocessors platform. We evaluate designs based on these two algorithms.

All these designs are implemented in our MPI-2 implementation over InfiniBand, MVAPICH2 [27], which is based on MPICH2 developed by Argonne National Laboratory [3].

4.1 Design Efficient MPI-2 Passive One-Sided Communication

In this section we discuss the challenges in implementing efficient MPI-2 passive one-sided communication. Then in the following two sections, we will describe thread based designs and atomic operations based designs for passive one-sided communication.

Synchronization Performance: When the contention for synchronization functions is low, low synchronization overhead is important for overall performance. When the contention is high, low synchronization delay is also desirable. At the same time, the number of messages exchanged for synchronization functions should be small.

Independent Progress: In MPI-2 passive one-sided communication, target process does not make any MPI call to cooperate with origin process for communication or synchronization. Therefore, the implementation can not rely on calling progress engine in MPI functions, which is the strategy commonly used for two-sided communication.

Handle Concurrent Communication: In MPI-2, one-sided communication and two-sided communication may happen concurrently. The one-sided communication from different origin processes to disjoint windows at a target process may also happen concurrently. Hence we need to handle this situation in our design.

Non-blocking MPI_Win_lock(): When MPI_Win_lock() is called if the lock at the target process is held by other processes, lock can not be acquired until the lock is released. Non-blocking MPI_Win_lock allows the current process to continue without waiting for the lock. However, the lock must be acquired before the first communication operation takes effect.

Shared Lock and Exclusive Lock: In MPI-2, both shared lock and exclusive lock are supported.

Utilize RDMA for Data Transfer: As we concluded in the last chapter, Modern network interfaces, such as InfiniBand, offer RDMA based operations, which can be utilized for high performance data transfer in one-sided communication. Therefore, synchronization mechanisms must work correctly with RDMA based communication implementation.

In following sections, we address these issues and describe thread-based and atomic operation-based designs for MPI-2 passive mode synchronization.

4.2 Thread Based Designs

Thread based design is widely used in MPI implementations to support one-sided communication. In our MPI-2 implementation over InfiniBand, we use RDMA feature to transfer data, while we still use a thread running at target process to handle some special cases such as non-contiguous data transfer and accumulate function.

Therefore, we would like to examine whether we can use this thread to handle passive synchronization efficiently or not.

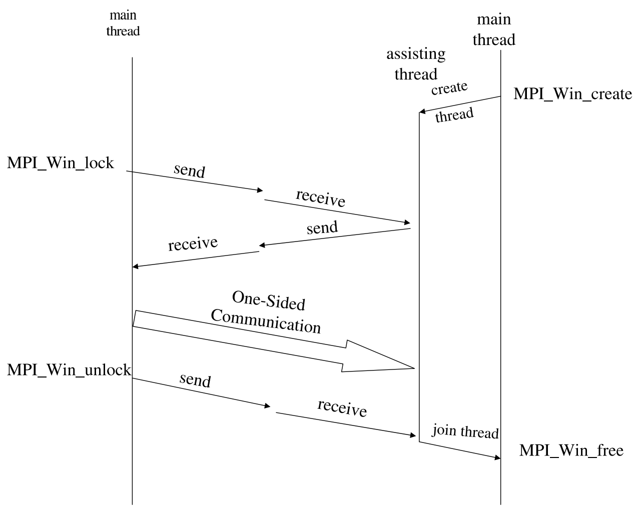

Figure 4.1: Dedicated Thread Based Design

4.2.1 Dedicated Thread Based Design

As shown in Figure 4.1, in Dedicated Thread Based design, an assisting thread runs at the target side in a dedicated manner, and handles all passive synchronization requests from origin processes. To achieve low latency, the thread is always active and uses polling to process communication. The characteristics of the thread guarantees the independent progress of synchronization process. Since before using RDMA operations to transfer data, we need to know when the lock is acquired, a round-trip time is used to acquire a lock, and another control message is used to release the lock. To implement non-blocking MPI_Win_lock(), after starting the lock request, origin process continues its work. Since in thread based design, target process knows whether the request message is for shared lock or exclusive lock, target process can coordinate between shared lock requests and exclusive lock requests to support both of them. In this implementation, passive synchronization messages and two sided messages are handled by different threads. Therefore, mutual exclusion mechanisms are needed before MPICH2 eventually becomes thread safe.

The dedicated thread based design can achieve good portability, because it is based on two-sided message passing, which is supported by all platforms. However, there are some shortcomings in its performance. One problem is related to concurrent communication. In addition to concurrent two-sided communication, one-sided communication can also happen concurrently with one-sided communication. When there are multiple windows at a target process, different origin processes may access them, hence using a single thread does not allow concurrent accesses. A simple solution is to use multiple threads, one thread per window. However, in the thread based designs, as long as all the threads are sharing the same progress engine, the time spent on the progress engine can not be overlapped, due to locking. Therefore, compared with the dedicated thread based design, using multiple threads can not improve performance. Performance of using multiple threads will be shown in Section 4.4. Another problem is that since the dedicated thread keeps running at a target process, CPU cycles are consumed by the thread even if there is no passive one-sided communication. In the next section, we introduce event driven based design to solve this problem.

4.2.2 Event Driven Based Design

As we showed in last chapter, we can use event driven feature to optimize accumulation operation. Here we propose an event driven based design to reduce the CPU utilization of thread. At the very beginning, a pre-defined event handler which can resume a blocked thread, is registered to HCA. When the assisting thread is created, the process posts a receive operation and blocks the thread. If a process wants to communicate with a remote process, it first posts a send operation matching with the pre-post receive operation at the remote party, to generate a signal. Then the event will wake up the thread. The remaining steps are similar to dedicated thread based design. Finally, before the thread is blocked again, another receive operation will be posted.

In the event driven based design, a thread can be blocked while idle to reduce the CPU utilization. However, compared with the dedicated thread based design, extra time is spent on generating the signal and waking up the thread. Performance numbers of the event driven based design will be shown in section 4.4.

4.3 Atomic Operation Based Design

As we described, hardware level remote atomic operations are supported in InfiniBand. This gives us the opportunity to exploit some well known algorithms proposed for shared-memory synchronization. In this section, we present two designs based on "Test&Set" and MCS lock algorithms.

By using hardware level atomic operations, there is no control messages between the origin process and the target process for the passive synchronization functions any more. Therefore, the assisting thread at the target process is not needed to make progress for the communication of control messages.

Eliminating the assisting thread can significantly reduce CPU utilization, while we also lose the support of send/receive based one-sided communication, which is relying on the assisting thread to make progress. Our solution is to use the RDMA based designs, which has been introduced in the last chapter. In some special cases, such as transferring data with datatype, the assisting thread could be beneficial. This would be a direction to explore in the future.

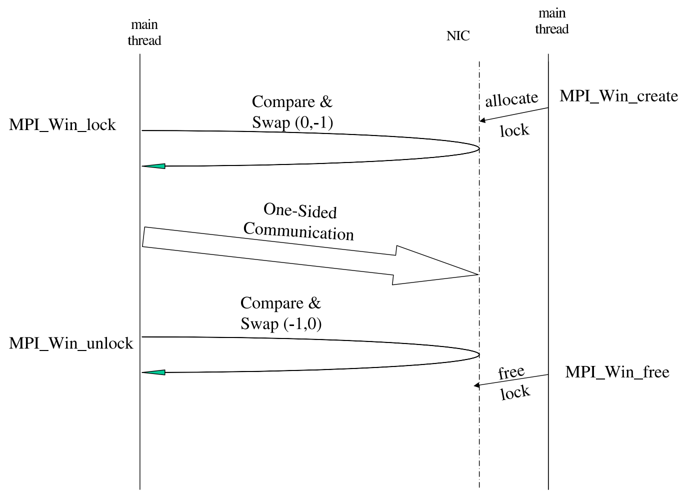

Figure 4.2: Test and Set" Algorithm

4.3.1 Test&Set Based Design

In this algorithm, a flag is used to indicate whether the lock is held or not. As shown in Figure 4.2, to acquire a lock, a processor makes effort to change the flag from false to true by executing a "Test and Set" instruction. A processor releases the lock by changing the flag back to true. To build up the MPI-2 synchronization functions with "Test and Set" lock algorithm, we can use the atomic operation Compare-and-Swap.

Since the atomic operations are handled by HCA and the MPI library at target process is not involved, the progress of passive synchronization is independent of the progress of the target process. To utilize RDMA for data transfer, when the Compare-and-Swap for acquiring lock succeeds, the process can start using RDMA to transfer data. To implement nonblocking MPI_Win_lock, a process only issues the first Compare-and-Swap operation in MPI_Win_lock(), without waiting for its success. The waiting is delayed until the first communication operation.

We can easily extend Test&Set algorithm to support both shared lock and exclusive lock, by checking the value returned by Compare-and-Swap operation. To request an exclusive lock, a process uses one or multiple Compare-and-Swap operations with compared_value 0 and swap_value -1 to get the lock. To requests a shared lock, a process uses one Compare-and-Swap operation with compared_value 0 and swap_value 1. If it succeeds, the process acquires the lock. Otherwise, following Compare-and-Swap operations will be issued with compared_value equal to the last returned value and swap_value equal to that of the value plus one. To release the control, a process can use a Fetch-and-Add operation to increase(exclusive lock) or decrease(shared lock) the value of the flag.

The problem of "Test and Set" based design is the high network traffic caused by repeated issue of Compare-and-Swap operations. Using an exponential back off mechanism can alleviate this problem.

4.3.2 MCS Based Design

MCS is proposed as a scalable synchronization algorithm for shared memory multiprocessors platform [25]. The main idea of MCS is to maintain a distributed queue

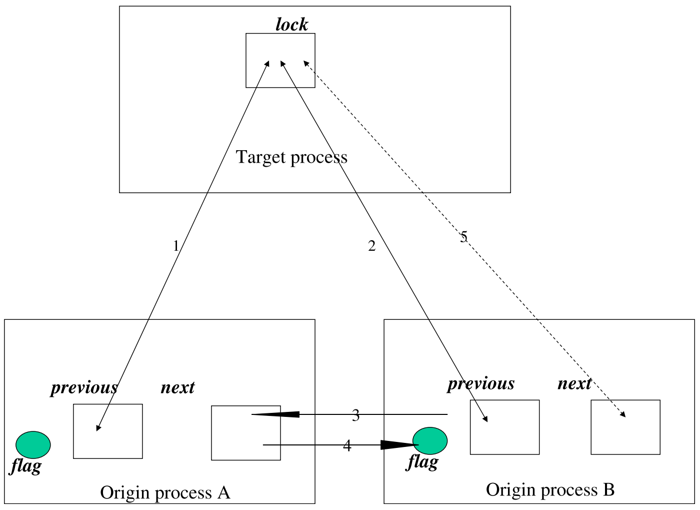

Figure 4.3: MCS Algorithm

for processes that are competing for the lock. Scalability is achieved by avoiding spinning on the remote memory.

For each window, each origin process maintains three data structures: *flag*, *previous* and *next*, and each target process maintains one data structure, called *lock*. When an origin process A requests for the lock on the target process, it swaps its process id with the value of *lock*,which is indicated by line 1 in Figure 4.3. Then an origin process B requests for the same lock by swapping (indicated by line 2). Based on the value swapped back, the origin process B knows that the origin process A is queued before it. Thus, it updates the value of *next* in origin process A (indicated by line 3). When the origin process A releases the lock, basing on the value of *next*, origin process A updates the value of *flag* in the origin process B (indicated by line 4). Finally, when the origin process B releases the lock, it resets *lock* at target process to null (indicated by line 5). We use Compare-and-Swap to update the data with atomic requirement and use RDMA Write to update the data without this requirement.

Compared with other atomic operation based synchronization algorithms, MCS has advantage in terms of scalability. However, since InfiniBand does not support atomic Swap operation, emulating it with Compare-and-Swap operations will lose its advantage. In Section4.4, we will show that when there are multiple processes competing for the same lock, the number of messages exchanged in MCS based design is no longer constant.

4.4 Performance Evaluation

The performance of MPI-2 passive synchronization functions can be evaluated with respect to the following metrics.

1) Synchronization Overhead: Time spent on synchronization functions.

2) Synchronization Delay: Amount of time required after one origin process releases a lock on a remote window and another origin process acquires the same lock.

3) Concurrency: The capability to handle multiple concurrent passive synchronization functions.

4) Message Complexity: The number of messages exchanged for synchronization functions.

5) CPU Utilization: The CPU cycles involved in the synchronization process. The relationship between synchronization performance and CPU cycles used.

4.4.1 Experimental Testbed

Our experimental testbed consists of a cluster of 8 SuperMicro SUPER X5DL8-GG nodes, each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X

64-bit 133 MHz bus, and connected to Mellanox InfiniHost MT23108 DualPort 4x HCAs. The nodes are connected using the Mellanox InfiniScale 24 port switch MTS 2400. The kernel version used is Linux 2.4.22smp. The InfiniHost SDK version is 3.0.1 and HCA firmware version is 3.0.1. The Front Side Bus (FSB) runs at 533MHz. The physical memory is 1 GB of PC2100 DDR-SDRAM memory.

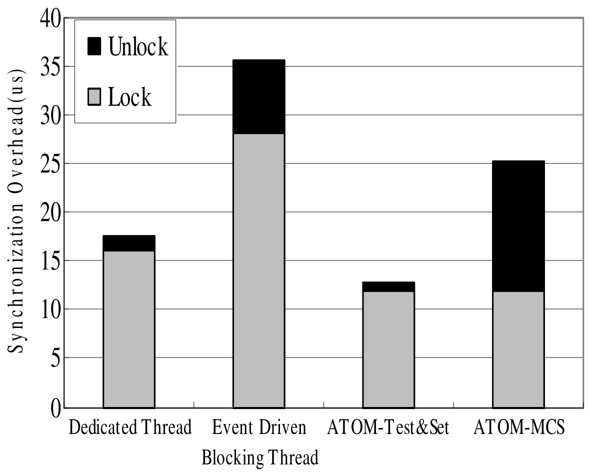

Figure 4.4: Synchronization Overhead

4.4.2 Synchronization Overhead

In this subsection, we use a simple approach to measure synchronization overhead. In this test, one process calls only MPI-2 passive synchronization functions (MPI_Win_lock and MPI_Win_unlock) on a window at the other process for multiple iterations. We then report the time taken for each iteration. Figure 4.4 shows the synchronization overhead for all the four designs. We also report time spent on lock acquiring and time spent on lock releasing separately. We can see that "Test and Set" based design shows the best performance with around 12.83 microseconds. Releasing a lock is cheaper for this design because processes do not need to wait for the completion of unlock. We also see that using a dedicated thread can achieve better performance than using atomic operations with MCS. The event driven approach shows the worst performance.

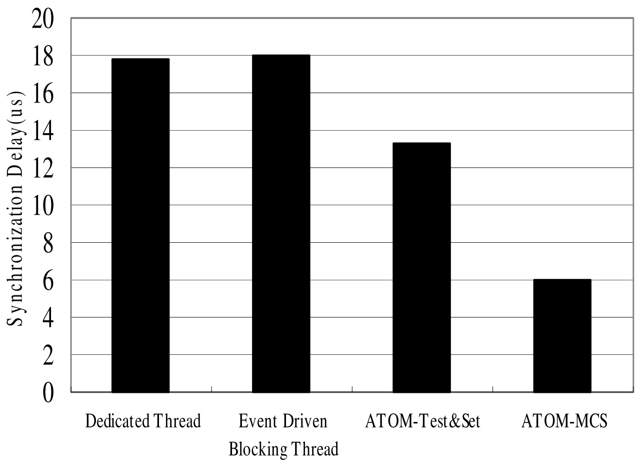

Figure 4.5: Synchronization Delay

4.4.3 Synchronization Delay

Synchronization Delay is the delay between one origin process releasing a lock on a remote window and another origin process acquiring the same lock. It is an important performance metric for lock algorithm, especially when the competition between different origin processes for a given lock is heavy. The test for measuring synchronization delay consists of multiple iterations, using two origin processes and one target process. In the even numbered iterations, origin process 1 requests lock earlier than origin process 2, and in the odd numbered iterations, origin process 2 requests lock earlier than origin process 1. After acquiring the lock, each process holds the lock for time E, and then releases the lock. The value of E we used is always longer than the synchronization overhead of all designs. As we can see in Figure 4.5, the designs based on atomic operations out performs thread based designs. The MCS based design shows the best synchronization delay because locks can be transferred to the next process using a single message. In all other designs, at least a round-trip time is required.

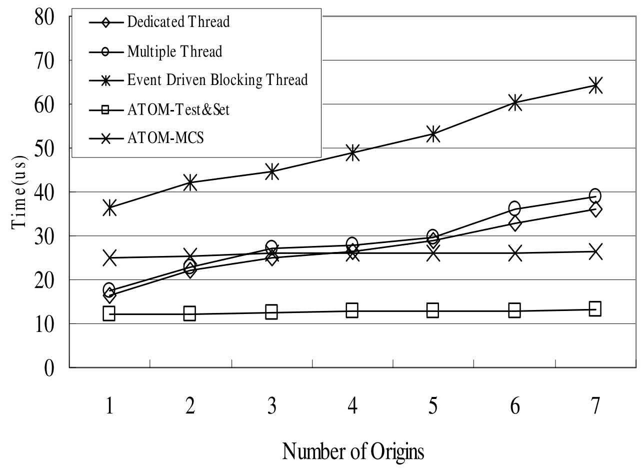

Figure 4.6: Concurrency

4.4.4 Concurrency

For some MPI-2 applications, the target process may have a large volume of data to be accessed by multiple origin processes. One way to improve the application performance is use multiple windows and let different origin processes concurrently access the data in different windows. To evaluate how different designs handle concurrent accesses, we used a test with multiple origin processes and one target process. In the target process, multiple windows are created, and the number of windows is equal to the number of origin processes. In each iteration, each origin process calls only MPI_Win_lock and MPI_Win_unlock on the corresponding target window. We then report the average time spent on each iteration. Figure 4.6 shows that for Test&Set based design and MCS based design, the time spent on synchronization functions does not change. This indicates that they can handle concurrent accesses in an efficient manner. However, for thread based design, the time increases when the number of origin processes increases. Further, we can see that even by using multiple threads, we can not achieve better concurrency.

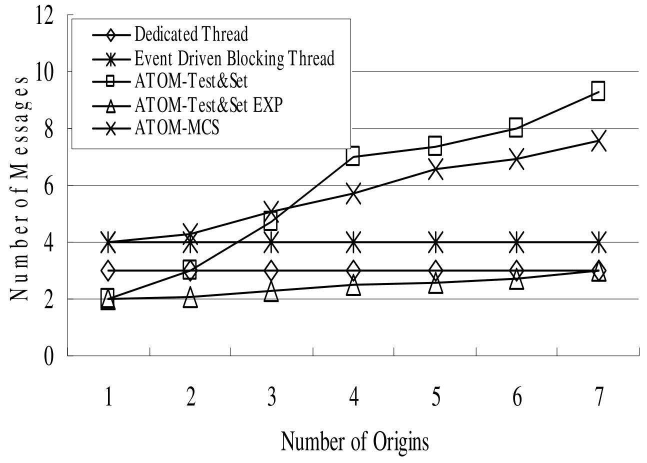

Figure 4.7: Message Complexity

4.4.5 Message Complexity

Message complexity shows the number of messages exchanged for synchronization functions and a good design should have low message complexity. The test uses multiple origin processes and one target process, and all the origin processes compete for the same lock on a target window. Then we calculate the average number of messages exchanged between one origin process and one target process.

As we can see in Figure 4.7, with increase of the number of origin processes, the number of messages of both thread based designs does not change. For Test&Set based design, exponential back off mechanism can reduce the number of messages from 9 to 3 when the number of origin processes is 7. Although MCS algorithm is proposed as a scalable algorithm, without SWAP operation support, the number of messages increases to around 7, when the number of origin processes is 7.

4.4.6 CPU Utilization

As we have mentioned, one shortcoming of the thread based designs is that the assisting thread consumes significant amount of CPU cycles. We evaluate this problem in two scenarios.

Computing Thread:

In MPI-2 application, each process may have multiple computing threads running in it. This test is to evaluate the performance of synchronization functions under this scenario. This test uses two processes: one origin process and one target process. The target process spawns several computing threads, and the origin process calls synchronization functions MPI_Win_lock and MPI_Win_lock on a window at the target process for multiple iterations. We then report the time for each iteration.

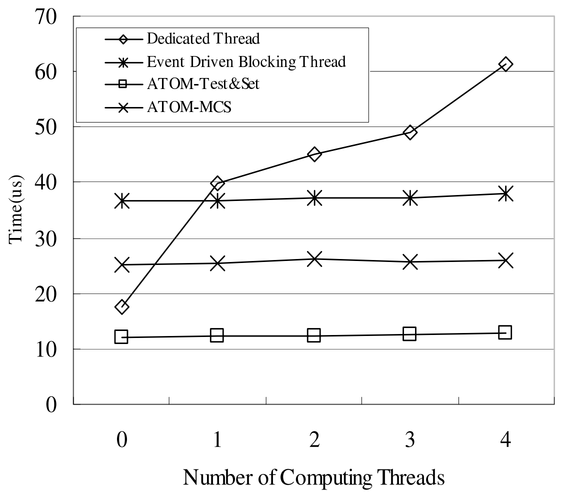

Figure 4.8: Computing Thread

Figure 4.8 shows that for both atomic based designs, the time remains almost unchanged, while for the dedicated thread based design, the time increases with the increase of the number of the computing threads. For the event driven based design, since the assisting thread is waken up by a signal, the time almost remains constant too.

SMP Mode:

Until now, all our evaluations are done under *UniProcessor* mode, which means we assign one MPI application process on each node. But many scientific applications are computation-intensive, and running them in *SMP* mode is more suitable. To evaluate this aspect, we proposed a test which is based on synchronization overhead test we used before. The *SMP* version test allows for running two origin processes (A and B) on one dual-CPU workstation, and running two target processes (C and D) on the other dual-CPU workstation. The origin process A calls synchronization functions to

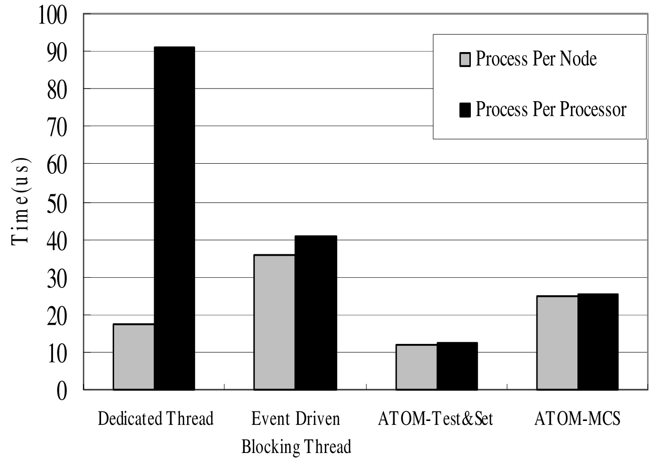

Figure 4.9: Assignment Mode

the target process C, while the origin process B does the same thing to the target process D. Figure 4.9 shows that, both atomic operation based designs can deliver similar performance in both modes. However, the dedicated thread based design does not work well in *one process per processor* mode due to the polling of communication threads.

4.4.7 Discussion

From the performance results we can see that in general, atomic operation based designs outperform thread based designs. By taking advantage of atomic operations in InfiniBand, we can achieve better synchronization overhead and synchronization delay. Atomic operation based designs can also achieve better concurrency and independent communication progress. One possible drawback of atomic operation based design is that the number of messages to acquire a lock increases when there is high contention for the lock. For Test&Set based design, this problem can be solved by using exponential backoff. For MCS based design, the problem is due to the lack of an atomic SWAP operation in the current InfiniBand implementation.

4.5 Summary

In this chapter, we analyzed issues and concerns related to designing a high performance MPI-2 passive synchronization mechanisms on InfiniBand clusters. We proposed, implemented and evaluated two thread based designs (i.e., Dedicated Thread and Event Driven Blocking Thread based designs) and two atomic operation based designs (i.e., Test&Set and MCS based design). We demonstrated that by taking advantage of InfiniBand atomic operations, we can achieve efficient synchronization and deliver good performance.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis, we proposed some novel designs for MPI-2 one-sided communication model. We made an attempt to leverage the fast and scalable primitives offered by the IBA software and hardware to improve the performance of the one-sided data transfer and synchronization functions.

Many existing MPI-2 one-sided communication implementations are built on top of MPI send/receive operations. Although this approach can achieve good portability, it suffers from high communication overhead and dependency on remote process for communication progress. To address these problems, we propose a high performance MPI-2 one-sided communication design over the InfiniBand Architecture. In our design, MPI-2 one-sided communication operations such as MPI_Put, MPI_Get and MPI_Accumulate are directly mapped to InfiniBand Remote Direct Memory Access (RDMA) operations. Our design has been implemented based on MPICH2 over InfiniBand. We present detailed design issues for this approach and perform a set of micro-benchmarks to characterize different aspects of its performance. Our performance evaluation shows that compared with the design based on MPI send/receive, our design can improve throughput up to 77%, and reduce lantency and synchronization overhead up to 19% and 13%, respectively. Under certain process skew, the bad impact can be significantly reduced by new design, from 41% to nearly 0%. It also can achieve better overlap of communication and computation.

Further, we compare various design alternatives for passive synchronization in MPI-2 one-sided communication on InfiniBand clusters. We discuss several requirements for synchronization in passive one-sided communication. Based on these requirements, we present four design alternatives, which can be classified into two categories: thread-based and atomic operation based. In thread-based designs, synchronization is achieved with the help of extra threads. In atomic operation based designs, we exploit InfiniBand atomic operations such as Compare-and-Swap and Fetch-and-Add. Our performance evaluation results show that the atomic operation based design can achieve less synchronization overhead, better concurrency, and less computing resource consumption comparing with the thread based design.

In future, we plan to use real applications to study the impact of our InfiniBand enabled approach. We also plan to investigate how to handle datatype efficiently in MPI-2 one-sided communication.

BIBLIOGRAPHY

[1] 10 Gigabit Ethernet Alliance. http://www.10gea.org/.

[2] Quadrics Supercomputers World Ltd. http://www.quadrics.com/.

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

[4] Noboru Asai, Thomas Kentemich, and Pierre Lagier. MPI-2 Implementation on Fujitsu Generic Message Passing Kernel. In *SC*, 1999.

[5] InfiniBand Trade Association. http://www.infinibandta.org.

[6] M. Banikazemi, B. Abali, L. Herger, and D. K. Panda. Design Alternatives for VIA and an Implementation on IBM Netfinity NT Cluster. *Special Issue of the Journal of Parallel and Distributed Computing (JPDC), Vol. 61, No. 11, pp. 1512-1545*, November 2001.

[7] Massimo Bertozzi, Marco Panella, and Monica Reggiani. Design of a VIA Based Communication Protocol for LAM/MPI Suite. In *9th Euromicro Workshop on Parallel and Distributed Processing*, September 2001.

[8] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.

[9] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[10] Dan Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, Computer Science Division, University of California at Berkeley, October 2002.

[11] Stephen Booth and Fernando Elson Mourao. Single Sided MPI Implementations for SUN MPI. In *Supercomputing*, 2000.

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

[13] GigaNet Corporations. cLAN for Linux: Software Users' Guide.

[14] GigaNet Corporations. http://www.giganet.com.

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

[16] Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra. Evaluating the Performance of MPI-2 Dynamic Communicators and One-Sided Communication. In *EuroPVM/MPI*, September 2003.

[17] M. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas. Portable and Effcient Parallel Computing Using the BSP Model. *IEEE Transactions on Computers*, pages 670–689, 1999.

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

[19] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA, USA, 1998.

[20] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[21] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. 12th IPPS.

[22] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[23] M. Tomasevic J. Protic and V. Milutinovic. Distributed shared memory: Concepts and systems. In *IEEE Parallel & Distributed Technology, 4(2):63–79*, Summer 1996.

[24] J. Traff and H. Ritzdorf and R. Hempel. The Implementation of MPI-2 One-Sided Communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.

[25] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. In *ACM Transactions on Computer System*, 1991.

[26] LAM Team, Indiana University. LAM 7.0.4.

[27] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In proceeding of IPDPS, April 2004.

[28] Mitchell L. Loeb, Andrew J. Rindos, William G. Holland, and Steven P. Woolet. Gigabit Ethernet PCI Adapter Performance.

[29] Maciej Golebiewski and Jesper Larsson Traff. MPI-2 One-Sided Communications on a Giganet SMP Cluster. In *EuroPVM/MPI*, 2001.

[30] Martin Schulz. Efficient Coherency and Synchronization Management in SCI based DSM systems. In *SCI-Europe, conference stream of Euro-Par*, 2000.

[31] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[32] Message Passing Interface Forum. MPI-2: A Message Passing Interface Standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[33] Elson Mourao and Stephen Booth. Single Sided Communications in Multiprotocol MPI. In *EuroPVM/MPI*, 2000.

[34] Fernando Elson Mourao and J Gabriel Silva. Implementing MPI's One-Sided Communications for WMPI. In *EuroPVM/MPI*, September 1999.

[35] Myricom. GM Messaging Software. http://www.myri.com/scs/index.html.

[36] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, 1586, 1999.

[37] OSU Network-Based Computing Laboratory. MPI over InfiniBand Project. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html.

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

[39] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *the Proceedings of the IEEE International Conference on Hot Interconnects*, August 2001.

[40] Quadrics Supercomputers World Ltd. Elan Reference Manual. 1999.

[41] W. Rehm, Ren Grabner, Frank Mietke, Torsten Mehlan, and Christian Siebert. Development of an MPICH2-CH3 Device for InfiniBand. http://www.tu-chemnitz.de/informatik/RA/cocgrid-/Infiniband/pmwiki.php/InfiniBandProject/ProjectPage.

[42] Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet Message Passing. In *the Proceedings of the IEEE International Conference on Supercomputing*, pages 57–64, Denver, Colorado, November 10-16 2001.

[43] Piyush Shivam, Pete Wyckoff, and Dhabaleswar K. Panda. Can User-Level Protocols Take Advantage of Multi-CPU NICs? In *the Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 15-19 2002.

[44] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra. *MPI–The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition.* The MIT Press, 1998.

[45] V. Buch T.Eicjen, A. Basu and W. Vogels. U-Net:A User-Level Network Interface for Parallel and Distributed Computing. In *SIGOPS*, 12 95.

[46] Top 500 Supercomputers website. http://www.top500.org.

[47] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[48] Joachim Worringen, Andreas Gaer, and Frank Reker. Exploiting Transparent Remote Memory Access for Non-Contiguous and One-Sided-Communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.