DESIGNING SCALABLE AND HIGH PERFORMANCE ONE SIDED COMMUNICATION MIDDLEWARE FOR MODERN INTERCONNECTS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Gopalakrishnan Santhanaraman, M. S

* * * * *

The Ohio State University

2009

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. F. Qin

Dr. P. Balaji

Approved by

Adviser

Graduate Program in Computer Science and Engineering

ABSTRACT

High-end computing (HEC) systems are enabling scientists and engineers to tackle grand challenge problems in their respective domains and make significant contributions to their fields. Examples of such problems include astro-physics, earthquake analysis, weather prediction, nanoscience modeling, multiscale and multiphysics modeling, biological computations, computational fluid dynamics, etc. There has been great emphasis on designing, building and deploying ultra scale HEC systems to provide true petascale performance for these grand challenge problems. At the same time, Clusters built from commodity PCs are being predominantly used as main stream tools for high-end computing owing to their cost-effectiveness and easy availability.

Communication subsystem plays a pivotal role in achieving scalable performance in clusters. Of late there has been a lot of interest in one-sided communication model and they are seen as a viable option for petascale applications. The one-sided communication provides good potential for computation communication overlap. In order to provide high performance and scalability, the one-sided communication subsystem needs to be designed to leverage the advanced capabilities of the modern interconnects.

In this dissertation we study and explore various aspects of one-sided communication like zero-copy, overlap, reduced remote CPU utilization, latency hiding techniques,and non-contiguous data transfers in middleware libraries. We improved the passive synchronization design to use RDMA atomic operations that provides high overlap capability. We also proposed a hybrid design that extends the above approach to optimize intra-node communications as well. We have also explored the use of remote completion semantics for RDMA operations in InfiniBand to improve the performance of fence synchronization. To optimize non-contiguous data communication, we proposed novel zero-copy designs using InfiniBand scatter/gather operations with reduced remote CPU utilization. Designs using RDMA atomic primitives have been proposed to improve the performance of read-modify-write operations. Further we have also proposed latency hiding techniques that uses non-blocking semantics and aggregation mechanisms.

Dedicated to my parents

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for guiding me throughout the duration of my PhD study. I'm thankful for all the efforts he took for my dissertation.

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

I'm grateful for financial support by National Science Foundation (NSF) and Department of Energy (DOE). I'm thankful to Dr. Jarek Nieplocha, Dr. Rajeev Thakur and Dr. Pavan Balaji for their support and guidance during my summer internships.

I would like to thank all my nowlab collegues past and present. I am fortunate for having worked with Sundeep, Amith, Vishnu, Sayantan, Karthik, Ranjit, Jin, Pavan, Darius, Juixing, Jiesheng, Adam, Matt, Wei, Lei, Weikuan, Qi, Rahul, KGK, Hari, Jaidev, Tejus, Greg, Ping, Ouyang, Krishan, Sreeram and Jonathan.

I am especially grateful to my friends Sundeep, Amith, Mallu, Chala and Niranjan for all their support during my stay at OSU

Finally I would like to thank my parents, my brother Raj and my wife Harini for their constant support and encouragement. I would not have made it this far without them.

VITA

August 1999 - Dec 2001	M.S Computer Engineering, University of South Carolina.
August 1994 - July 1998	B.Tech Ceramic Engineering, Institute of Technology, Banaras Hindu University, varanasi, India.
June 2008 - September 2008	Summer Intern, Argonne National Laboratory, Chicago, IL.
June 2005 - September 2005	Summer Fellow, Pacific NorthWest National Laboratory, Richland, WA.

PUBLICATIONS

G. Santhanaraman, J. Wu, W. Huang, and D. K. Panda, "Designing Zero-copy MPI Derived Datatype Communication over InfiniBand: Alternative Approaches and Performance Evaluation" ,The special Issue of the International Journal of High Performance Computing Applications (IJHPCA).

V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, and D. K. Panda, "Optimization and Performance Evaluation of Mechanisms for Latency Tolerance in Remote Memory Access Communication on Clusters", The International Journal of High Performance Computing and Networking (IJHPCN).

K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop and D. K. Panda, " Designing Multi-Leader-Based Allgather Algorithms for Multi-Core Clusters ", In proceedings of Workshop on Communication Architecture for Clusters (CAC 09).

G. Santhanaraman, P. Balaji, K. Gopalakrishnan, R. Thakur, W. Gropp and D. K. Panda, "Natively Supporting True One-sided Communication in MPI on Multi-core

Systems with InfiniBand ", In proceedings of International Symposium on Cluster Computing and the Grid (CCGrid'09).

R. Kumar, A. Mamidala, M. Koop, G. Santhanaraman and D. K. Panda, "Lock free asynchronous rendezvous design for point to point communication ", In proceedings of Europvm MPI 2008.

G. Santhanaraman, S. Narravula, and D. K. Panda, "Designing Passive Synchronization for MPI-2 One-Sided Communication to Maximize Overlap ",IEEE International Parallel and Distributed Processing Symposium (IPDPS '08).

S. Narravula, A. R. Mamidala, A. Vishnu, G. Santhanaraman and D. K. Panda, "High Performance MPI over iWARP: Early Experiences ",International Conference on Parallel Processing (ICPP '07).

G. Santhanaraman, S. Narravula, A. R. Mamidala and D. K. Panda, "MPI-2 One Sided Usage and Implementation for Read Modify Write operations: A case study with HPCC ", In Proceedings of EuroPVM/MPI '07.

A. Mamidala, S. Narravula, A. Vishnu, G. Santhanaraman and D. K. Panda, "Using Connection-Oriented vs. Connection-Less Transport for Performance and Scalability of Collective and One-sided operations: Trade-offs and Impact ",ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '07).

W. Huang, G. Santhanaraman, H. -W. Jin, and D. K. Panda, "Design of High Performance MVAPICH2: MPI-2 over InfiniBand ", In Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'06).

W. Huang, G. Santhanaraman, H. -W. Jin, and D. K. Panda, " Design Alternatives and Performance Trade-offs for Implementing MPI-2 over InfiniBand ", In Proceedings of EuroPVM/MPI 2005.

A. Vishnu, G. Santhanaraman, W. Huang, H. -W. Jin and D. K. Panda, "Supporting MPI-2 One Sided Communication on Multi-Rail InfiniBand Clusters: Design Challenges and Performance Benefits ", In Proceedings of the International Conference on High Performance Computing(HiPC'05).

W. Huang, G. Santhanaraman, H. -W. Jin and D. K. Panda, "Scheduling of MPI-2 One Sided Operations over InfiniBand ", In Proceedings of Workshop on Communication Architecture on Clusters (CAC 05) .

G. Santhanaraman, J. Wu and D. K. Panda, "Zero-Copy MPI Derived Datatype Communication over InfiniBand ", In Proceedings of EuroPVM/MPI 2004.

V.Tipparaju, G. Santhanaraman, J. Nieplocha and D. K. Panda, "Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand ", In Proceedings of International Parallel and Distributed Processing Symposium (IPDPS 04).

V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman and D. K. Panda, " Optimizing Mechanisms for Latency Tolerance in Remote Memory Access Communication on Clusters ", In Proceedings of IEEE Cluster Computing 2003.

J. Nieplocha, V. Tipparaju, M. Krishnan, G. Santhanaraman, and D. K. Panda, " Exploiting Nonblocking Remote Access communication in Scientific benchmarks on Clusters ", In Proceedings of International Conference on High Performance Computing (HiPC 2003).

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Prof. D. K. Panda
Computer Networks	Prof. D. Xuan
Software Systems	Prof. S. Parthasarathy

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

High-end computing (HEC) systems are enabling scientists and engineers to tackle grand challenge problems in their respective domains and make significant contributions to their fields. Examples of such problems include astro-physics, earthquake analysis, weather prediction, nanoscience modeling, multiscale and multiphysics modeling, biological computations, computational fluid dynamics, etc. There has been great emphasis on designing, building and deploying ultra scale HEC systems to provide true petascale performance for these grand challenge problems. At the same time, Clusters built from commodity PCs are being predominantly used as main stream tools for high-end computing owing to their cost-effectiveness and easy availability. In fact, the top 500 list of supercomputers [61] feature large scale clusters delivering TFlops of computational power. The easy availability of low cost commodity PC's together with scalable and high performance interconnection networks is making Compute Clusters more affordable and cost effective. With the advent of multi-core architecture, each of the nodes are being equipped with multiple cores allowing for ultra-scale cluster sizes up to hundreds of thousands and even millions of cores by the next decade.

However, the performance that applications can achieve on such large-scale systems depends heavily on their ability to avoid synchronization with other processes, thus minimizing idleness caused by process skew. Towards this goal, scientific applications can use two models for minimizing such synchronization requirements—clique-based communication and implicit data movement using one-sided operations.

Clique-based communication refers to the ability of applications to form small sub-groups of processes with a majority of the communication happening within the groups. Nearest neighbor (e.g., PDE solvers, molecular dynamics simulations) and cartesian grids (e.g., FFT solvers) are popular examples of such communication [5, 23, 8]. While clique-based communication reduces the number of processes each process needs to synchronize with, it does not completely avoid synchronization. Similarly, while the size of the clique grows slowly as compared to the overall system size, on ultra-scale systems, this can still be a concern. For example, in a 2-D cartesian grid communication along a row of processes, on a million process system, each clique can contain as many as a thousand processes.

Implicit data movement using one-sided operations supplements the benefits of clique-based communication by allowing data to be moved from one process' memory to another without requiring any synchronization.

A majority of the scientific and engineering application codes use MPI as the programming model. MPI provides an easy and portable abstraction for exchanging data between processes. It provides for a plethora of communication operations with varying semantics and usage. The MPI-1 [30] standard provides communication semantics for two-sided operations (send and receive). It has support for both point-to-point and collective communications. The MPI-2 standard [43] added new

one-sided communication semantics with various operations (Put,Get) and synchronization semantics.

Most modern as well as legacy parallel programming models (e.g., MPI [31], UPC [2], Global Arrays [4]) are increasingly providing constructs for such one-sided communication also known as RMA (remote memory access), where a process can read/write data from another process without necessarily requiring participation from the remote process.

The one-sided communication model can ideally minimize the need for synchronization. Since the remote process need not be involved in the data movement, it can perform its computation while the data transfer is happening. Thus this can lead to good potential for computation/communication overlap for the application.

However, in spite of these potential benefits, the adoption of these one-sided semantics in scientific applications has been slow. This has been primarily due to two reasons: (i) most legacy applications have been written using two-sided MPI semantics and many times, writing these applications in one-sided semantics may need changes to the algorithm and (ii) the one-sided designs are often implemented on top of two-sided semantics leading to poor performance.

Of late there has been a lot of interest in one-sided communication models and with modern interconnects providing better hardware support for RMA capabilities, they are seen as a viable option for petascale applications.

Recently InfiniBand Architecture (IBA)[35], a new industry proposed standard is making headway in the high performance networking domain. In addition to delivering low latencies and high bandwidth, it provides a rich set of network primitives like Remote Direct Memory Operations (RDMA), Remote Atomics, Scatter/Gather,

hardware-level Multicast and Send/Shared-Receive Queue capabilities. Also, the IBA standard allows for four conduits of of message transport, Reliable Connection (RC), Unreliable Connection, Reliable Datagram (RD) and Unreliable Datagram (UD) over which these network primitives can be layered. The RDMA capabilities of InfiniBand provides a good match to the one-sided RMA semantics.

The main objective of this dissertation is to design a High Performance and Scalable One sided Communication subsystem in MPI for the next-generation HEC systems. Such a system would exhibit good performance scaling while effectively harnessing the primitives exposed by the underlying high performance interconnect. In particular, we aim to address the following questions in the dissertation:

- How can we leverage the mechanisms of modern interconnects to build scalable and high performance one-sided communication and synchronization primitives?

- How can communication and synchronization mechanisms be redesigned to enable high overlap of computation and communication?

- What are the challenges associated in optimizing non-contiguous data communication and can the designs benefit from InfiniBand hardware support?

- Can we improve the performance of one-sided communication by designing non-blocking semantics and using techniques like re-ordering and aggregation?

- As the number of cores within a node increases, what kind of intra-node optimizations can one-sided communication benefit from?

The objectives described above all involve multiple challenges in terms of performance, scalability and ease of use. In this dissertation we study and investigate all these challenges to design an efficient and scalable one-sided communication subsystem that can provide benefits to applications.

The rest of this dissertation is organized as follows: In Chapter 2 we discuss existing technologies which provide background for our work including InfiniBand, multicore architecture, MPI, and details of one-sided communication middleware. Chapter 3 describes in detail the problems that are addressed in this dissertation. Chapters 4-10 discuss the detailed approaches and results for these problems. The significance and impact of the work in terms of open-source software developed as part of this dissertation is described in Chapter 11. Chapter 12 provides the conclusion and possible future research directions

CHAPTER 2

BACKGROUND

In this section we provide an overview of the InfiniBand Architecture and its features. Specifically, we explain the different communication semantics provided by IBA and the associated transports on which these are based on. Then we give a brief overview of multi-core architecture. Further, we also explain briefly the design overview of MVAPICH2 which is a popular MPI over InfiniBand and a brief overview of ARMCI which is another one-sided communication library.

2.1 InfiniBand Architecture Overview

InfiniBand Architecture (IBA) [35] is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. As shown in Figure 2.1, a typical IBA cluster consists of switched serial links for interconnecting processing nodes and the I/O nodes. The processing nodes are connected to the fabric by Host Channel Adapters(HCA). HCA's semantic interface to to the consumers is specified in the form of IB Verbs. The interface presented by Channel Adapters to consumers belongs to the transport layer. A queue-pair based model is used in this interface. Each Queue Pair is a communication endpoint. This can be seen in Figure 2.2. A Queue Pair consists of a send queue and a receive queue. Two

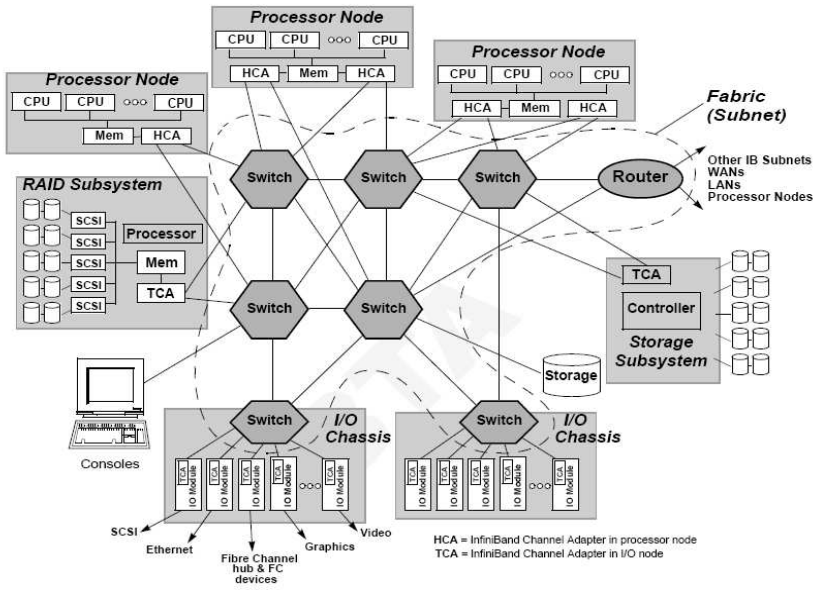

Figure 2.1: InfiniBand Architecture (Courtesy IBTA)

QPs on different nodes can be connected to each other to form a logical bi-directional communication channel. An application can have multiple QPs. Communication requests are initiated by posting descriptors (WQRs) to these queues. InfiniBand supports different classes of transport services. These are explained in the following section.

2.1.1 Send/Recv and RDMA

IBA supports two types of communication primitives: Send/Recv with Channel Semantics and RDMA with Memory Semantics. In Channel semantics, each send request has a corresponding receive request at the remote end. Thus there is a one-to-one correspondence between every send and receive operation. Receive operations require buffers posted on each of the communicating QP, which amount to a large number. In order to allow sharing of communication buffers, IBA allows the use of

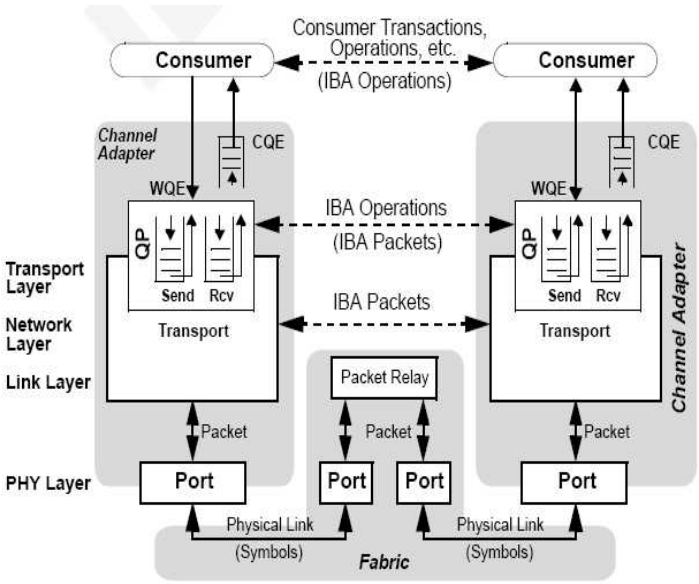

Figure 2.2: InfiniBand Protocol Stack (Courtesy IBTA)

Shared Receive Queues (SRQ). SRQs allow multiple QPs to have a common Receive Queue. In memory semantics, Remote Direct Memory Access (RDMA) operations are used. These operations do not require a receive descriptor at the remote end and are transparent to it. For RDMA, the send request itself contains the virtual addresses for both the local transmit buffer and the receive buffer on the remote end. The RDMA operations are available with the RC Transport. These RDMA operations are a good match for one-sided operations since the receiver side can be transparent to the operation.

Figure 2.3 shows the basic working of both the RDMA and the Send/Recv models. The main steps involved are labeled with sequence numbers. The main difference between the two is the requirement of posting a receive descriptor for the send/recv

model. In addition to these, InfiniBand also provides *RDMA Write with Immediate* operations which offers the flexibility of providing notification that the data has reached the memory in addition to directly placing the data in the remote memory.

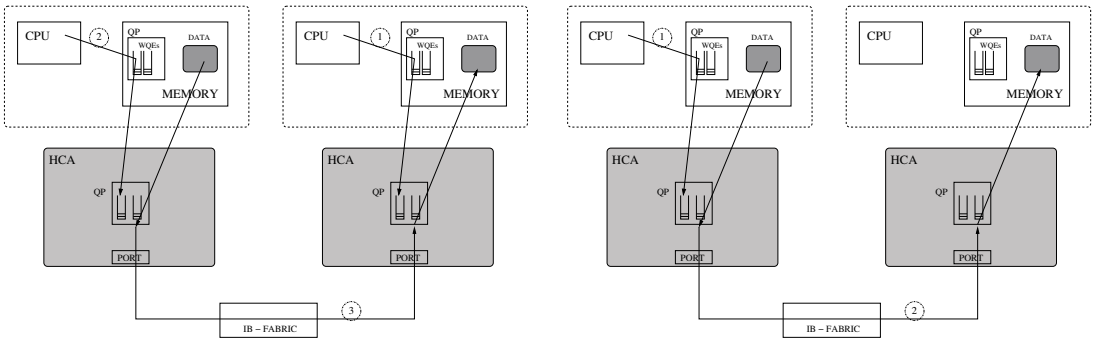

Figure 2.3: InfiniBand Transport Models: (a) Send/Recv Model and (b) RDMA Model

2.1.2 InfiniBand Scatter/Gather Capabilities

InfiniBand also provides Scatter/Gather capabilities to certain extent. In channel semantics, the sender can gather data from multiple locations in one operation. Similarly, the receiver can receive data into multiple locations. In memory semantics, non-contiguity is allowed only in one side. InfiniBand provides *RDMA Read with Gather* and *RDMA Write with Scatter* feature. RDMA Write can gather multiple data segments together and write all data into a contiguous buffer on the remote node in one single operation. RDMA Read can scatter data into multiple local buffers from a contiguous buffer on the remote node.

2.1.3 Hardware Remote Atomics in InfiniBand

One of the notable features provided by the InfiniBand Architecture is hardware atomic support. InfiniBand provides two network level remote atomic operations, namely, *fetch_and_add* and *compare_and_swap*. The network interface card (NIC) on the remote node guarantees the atomicity of these operations. These operations act on 64-bit values. In the atomic *fetch_and_add* operation, the issuing process specifies the value that needs to be added and the remote address of the 64-bit location to which this value is to be added. On the other hand, in an atomic *compare_and_swap* operation, the issuing process specifies a 'compare value' and a 'new value'. The value at the remote location is atomically compared with the 'compare value' specified by the issuing process. If both the values are equal, the original remote value is swapped with the new value which is also provided by the issuing process. If these values are not the same, swapping does not take place. In both the cases, the original value is returned to the issuing process. It is to be noted that these operations are atomic only with respect to other InfiniBand atomic operations.

2.2 Multicore architecture

Emerging trends in processor technology has led to Multicore Processors (also known as Chip-level Multiprocessing or CMP) which provides large number of cores on a single node thus increasing the processing capabilities of current generation systems. Dual-core (two cores per die) and Quad-core (four cores per die) architectures are widely available from various industry leaders including Intel, AMD, Sun (up to 8 cores) and IBM. the negligible cost associated with placing an extra processing core on the same die has allowed these architectures to increase the capabilities of the

applications significantly. Recently, Intel has announced that it will be introducing an 80-core die [6] within the next few years. Other industries are expected to follow this trend. Most HPC platforms are multi-core based in order to provide peta scale level computing. This brings an interesting trend that lots of communication can now happen within a node.

2.3 MPI Overview

Message Passing Interface (MPI) [55] was proposed as a standard communication interface for parallel applications. It specifies an API and its mapping to different programming languages such as Fortran, C and C++. Since its introduction, MPI has been implemented in many different systems and has become the *de facto* standard for writing parallel applications. The main communication paradigm defined in MPI is message passing. However, MPI is also implemented in systems that supports shared memory [29, 34]. Therefore, parallel applications written with MPI are highly portable. They can be used in different systems as long as there are MPI implementations available.

2.3.1 MPI Point-to-point Communication

In an MPI program, two processes can communicate using MPI point-to-point communication functions. One process initiates the communication by using MPI_Send function. The other process receives this message by issuing MPI_Recv function. Destination processes need to be specified in both functions. In addition, both sides specify a *tag*. A send function and a receive function match only if they have compatible tags.

MPI_Send and MPI_Recv are the most frequently used MPI point-to-point functions. However, they have many variations. MPI point-to-point communication supports different *modes* for send and receive. The mode used in MPI_Send and MPI_Recv is called *standard* mode. There are other MPI functions that support other modes such as *synchronous, buffered* and *ready* modes. Communication buffers specified in MPI_Send and MPI_Recv must be contiguous. However, there are also variations of MPI_Send and MPI_Recv functions that supports non-contiguous buffers. Finally, any send or receive functions in MPI can be divided into two parts: one to initiate the operation and the other one to finish the operation. These functions are called *non-blocking* MPI functions. For example, MPI_Send function can be replaced with two functions: MPI_Isend and MPI_Wait. By using MPI non-blocking functions, MPI programmers can potentially overlap communication with computation, and therefore increase performance of MPI applications.

2.3.2 MPI One-sided Communication

The MPI one-sided communication model is also known as the Remote Memory Access (RMA). In this model, a process defines a memory window in its local address space as the target for remote memory operations by other processes within the same MPI communicator. In one-sided communication, the origin process (the process that issues the RMA operation) can access a target process' remote address space also referred to as the window directly. In this model, the origin process provides all the parameters needed for accessing the memory area on the target process.

Data transfer happens through the one-sided operations: put, get and/or accumulate. In a put operation, the origin process writes data into the target's memory

window. In a get operation, the origin process reads data from the target's memory window to its local buffer. In an accumulate operation, the origin process can update atomically remote locations by combining the content of the local buffer with the remote memory buffer. Any of the predefined reduction operations like MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD, MPI_XOR, etc. can be performed. This one-sided operation combines communication and computation in a single interface.

To synchronize between the target (who provides the memory region) and the origin (who issues the data transfers) processes, MPI one-sided model defines both *active* and *passive* synchronization. Active synchronization involves both the origin and target processes and has either point-to-point semantics (post/start wait/complete) or collective semantics (fence). The post/start wait/complete mechanism allows only a subset of processes to synchronize. The fence has collective semantics that requires the participation of all processes in the group. Passive synchronization provides shared or exclusive lock semantics on the entire remote memory window and needs to involve only the origin process and the target process is uninvolved.

2.3.3 MPI Non-contiguous Data Communication

One of the important features provided by MPI is derived datatypes. MPI provides derived datatypes to enable users to describe noncontiguous memory layouts compactly and to use this compact representation in MPI communication functions. Derived datatypes also enable an MPI implementation to optimize the transfer of non-contiguous data. The MPI standard supports derived datatypes for both one-sided as well as two sided communication primitives.

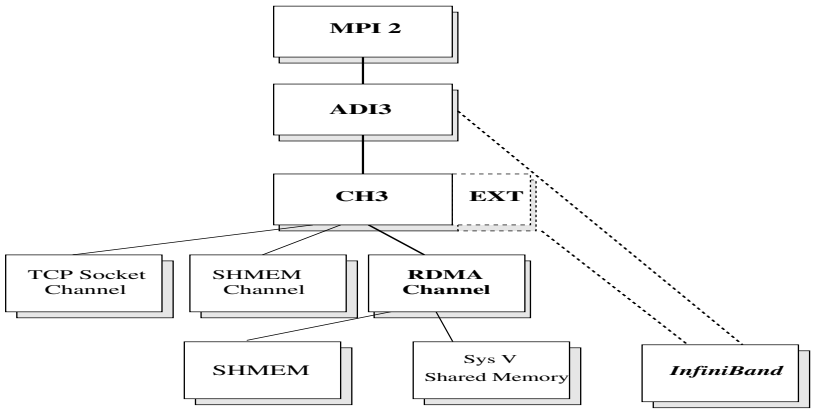

Figure 2.4: MVAPICH2 Design Overview

2.4 MVAPICH2 Overview

We now provide a high-level design overview of Point-to-Point and One-sided Communication support in the MVAPICH2 stack. MVAPICH2 [46] is a popular MPI over InfiniBand used worldwide. MVAPICH2 is an ADI3 level implementation on top of the MPICH2 stack. As a successor of MPICH, MPICH2 [9] supports MPI-1 as well as MPI-2 extensions including one-sided communication. In addition MVAPICH2 supports RDMA-based active one-sided communication by extending the CH3 layer as shown in Figure 2.4.

2.4.1 Point-to-point MPI Operations in MVAPICH2:

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

message. For intra-node communication, a separate shared memory channel is used for communication.

2.4.2 Point-to-point Based One-sided operations:

In MVAPICH2, all the one-sided operations discussed above in Section 2.3.2 are implemented over Point-to-Point operations. They are not optimal, but they are very portable. However when hardware support is available it is desirable to have a design that gives high performance and true one-sided communication.

2.4.3 Direct One-sided Operations:

As discussed above, one-sided operations implemented directly over the IBA can lead to significant performance gains. In fact, the basic get and put operations and active synchronization mechanisms are already implemented using RDMA Read and RDMA Write operations. The focus of this dissertation is to leverage mechanisms of RDMA , atomic operations and scatter gather support to provide optimized one-sided communication support in MVAPICH2.

2.5 ARMCI Overview

In addition to MPI, there are a few other libraries which provide one-sided programming model. Aggregate Remote Memory Copy Interface (ARMCI) [47] is a portable RMA communication library compatible with message-passing libraries such as MPI or PVM. It has been used for implementing distributed array libraries such as Global Arrays [50], other communication libraries such as Generalized Portable SHMEM [1] or the portable Co-Array Fortran compiler [19] at Rice University. ARMCI offers the following set of functionality in the area of RMA communication: 1) data

transfer operations; 2) atomic operations; 3) memory management and synchronization operations; and 4) locks. In scientific computing, applications often require transfers of noncontiguous data that corresponds to fragments of multidimensional arrays, sparse matrices, or other more complex data structures. With remote memory communication APIs that support only contiguous data transfers, it is necessary to transfer noncontiguous data using multiple communication operations. This often leads to inefficient network utilization and involves increased overhead. ARMCI offers explicit noncontiguous data interfaces: strided and generalized I/O vector that allow description of the data layout so that it could, in principle, be transferred in a single message.

CHAPTER 3

PROBLEM STATEMENT AND METHODOLOGY

There has been a lot of interest and research being done in the field of one-sided communication models recently. The main advantage of using this kind of model is that it supports asynchronous communication. There is no need to synchronize in terms of matching send/recv for every communication. In a one-sided communication model ideally only one process is involved in the communication and can directly read or write from the address space of the target process. The remote or target process need not be involved in this communication and can perform computation simultaneously. This can potentially lead to better computation/communication overlap. The MPI-2 standard provides one-sided communication or remote access memory (RMA) semantics in addition to two-sided semantics. However the one-sided primitives are often implemented on top of two-sided primitives thus resulting in poor performance. Also some of the semantics are restrictive for an application writer to take advantage of these operations. This has resulted in slow adoption of these semantics in scientific applications. At the same time, several applications like PS-DNS [3], ENZO [25],

AWM-Olsen [68], mpiBlast [20] have communication characteristics that can potentially benefit from one-sided communication model. Modern Interconnects like InfiniBand provide a lot of network features that are a close match for these one-sided or RMA operations.

The main objective of this dissertation is to *"Design a High Performance and Scalable One-sided Communication Subsystem leveraging directly the different network primitives of modern Interconnects for next-generation HPC systems"*.

Specifically, the dissertation aims to address the following challenges:

- Synchronization: Can we design truly one-sided passive synchronization using InfiniBand's hardware atomic operations to maximize overlap potential? How much of these overlap benefits can be translated to actual performance improvement in one-sided applications? Can existing design for collective synchronization like Fence be enhanced to improve scope for overlap as well as reduce any bottlenecks and hotspots in the network?

  In Chapter 4, we design support for passive synchronization using InfiniBand atomic operations. We also enhance the one-sided communication progress to provide scope for better overlap. In Chapter 6, we evaluate the different design options for implementing fence mechanism and propose a novel fence synchronization mechanism which uses InfiniBand's RDMA Write with Immediate capability to notify remote completions.

- Intra-node Optimizations for Multi-Core Architectures: Can we use the fast atomic locks provided by processor architectures for efficient intra-node locking?

What are the challenges to design support for fast CPU locks for intra-node operations and network based locks for remote operations?

In Chapter 5 we study the benefits of using fast CPU based locks for intra-node operations. We come up with a hybrid design that can migrate between CPU based locks and network locks depending on the migration policies. We demonstrate the benefits of this design for different communication patterns.

- Read-Modify-Write Mechanisms: Read-Modify-Write operations are important for one-sided applications. The MPI one-sided semantics does not explicitly provide this interface. How can one-sided applications be written using existing one-sided interface for read-modify-write functionality? Can InfiniBand's features such as atomic operations be used to achieve high performance and scalability for read-modify-write operations? What kind of hardware/network mechanisms are needed to further optimize these operations?

  In Chapter 7, we study the HPCC Random Access benchmark which primarily uses read-modify-write operations. We evaluate different approaches of efficiently implementing these operations using MPI-2 one-sided communication semantics. We also propose an implementation of MPI Accumulate that can make use of InfiniBand hardware fetch and add operations that yields good performance.

- Non-Contiguous operations: In several applications the data communication are often non-contiguous. The generic approach to handle non-contiguity is to perform packing and unpacking of data into contiguous buffers. This requires heavy CPU involvement on the origin and target side to copy the data into

contiguous buffer and copying the data out of contiguous buffers. Can the scatter/gather capabilities be utilized to achieve zero copy cost as well as reduced remote CPU involvement for non-contiguous data transfers for both point-to-point and two-sided data transfers? What are the trade-offs involved and what kind of application benefits can be achieved?

In Chapter 8, we discuss the various challenges in designing non-contiguous data communication for both two-sided as well as one-sided communication. The main overhead for non-contiguous communication is the overhead of data copies on both the sender and receiver sides. We propose new zero-copy designs for implementing non-contiguous data movement using InfiniBand's hardware based scatter/gather capability.

- Non-Blocking primitives: In order to obtain good computation communication overlap, the RMA one-sided design should support efficient non-blocking operations. How can we implement efficient non-blocking primitives that provides good scope for computation communication overlap? What are the challenges and what are the associated benefits of providing non-blocking primitives?

  In Chapter 9 we discuss the design issues in implementing non-blocking one-sided operations. We also demonstrate the performance benefits of a non-blocking design over a blocking design.

- Re-ordering and scheduling: MPI-2 standard allows the actual communication for RMA operations happen at synchronization time and also allows re-ordering

of operations within an access epoch. Can we design schemes that take advantage of this flexibility to achieve latency and better network bandwidth utilization? Can these schemes show performance benefits for some communication patterns?

In Chapter 10 we propose designs that can take advantage of the re-ordering semantics to interleave, prioritize and aggregate the operations. We demonstrate the performance benefits of these approaches for different communication scenarios.

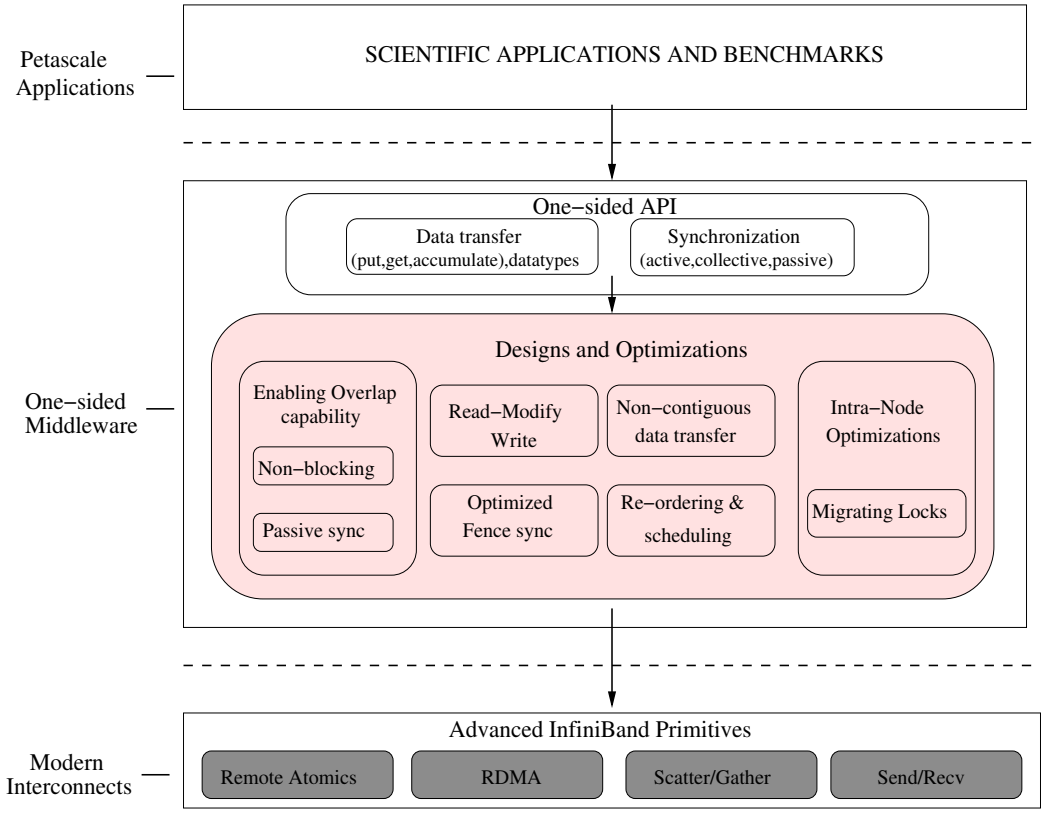

Figure 3.1: Broad Overview

Figure 4.1 provides an overview of all the above mentioned components. The components that we focus in this dissertation are lightly shaded. We describe the detailed design and the results of the various components of this dissertation in the following sections.

CHAPTER 4

PASSIVE SYNCHRONIZATION MECHANISM

The one-sided communication model decouples data transfer and synchronization operations. The synchronization operations ensure that the issued operations are complete and appropriate semantics are maintained. Depending on the type of synchronization, local and remote completions need to be ensured. These synchronization operations are very important in one-sided communication and it is very essential to provide efficient and low overhead synchronization mechanisms.

The MPI one-sided model provides two modes of synchronization.

- Active synchronization: where both the origin and target node are involved in the synchronization. It has both point-to-point semantics (post/start,wait/complete) as well as collective semantics (fence). The post/start wait/complete mechanism allows only a subset of processes to synchronize. The fence has collective semantics that requires the participation of all processes in the group.

- Passive synchronization: only the origin process is involved in the synchronization. In MPI-2 passive one-sided communication, the target process does not make any MPI calls to cooperate with the origin process for communication or synchronization. The synchronization is done through lock and unlock calls by the origin process on the window located on the target node.

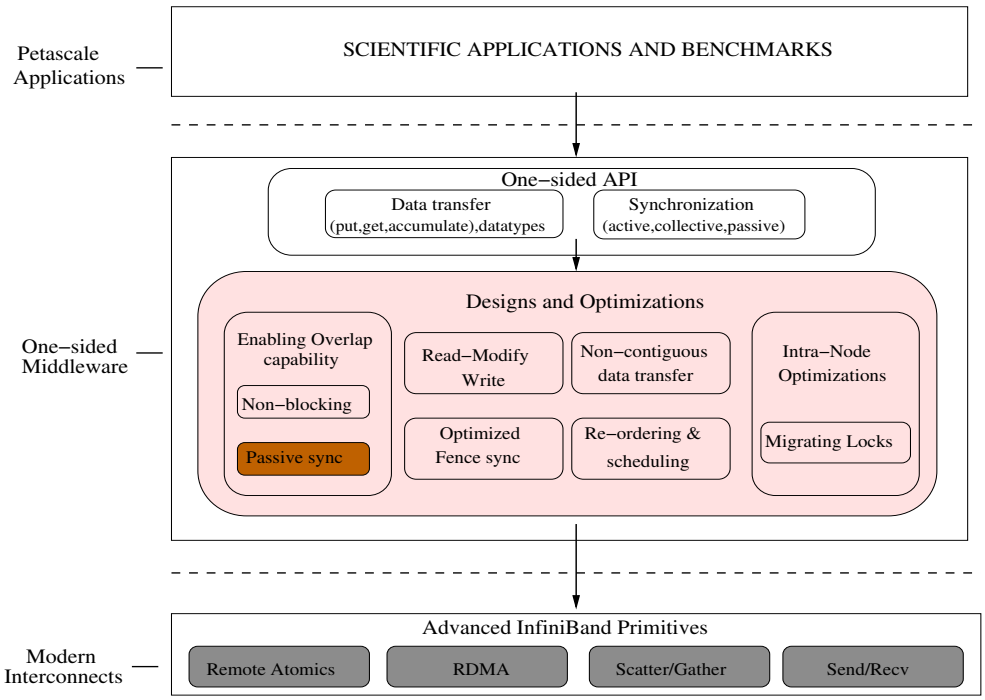

Figure 4.1: Overview

The passive synchronization offers true one-sided benefits. However most existing implementations do not provide these benefits because of limitations of current designs. In this chapter, we explain how the H/W atomic primitives can be leveraged for providing efficient and truly one-sided passive synchronzation mechanism which provides good overlap capability. Specifically we work on the highlighted part in Figure 4.1 of our proposed research framework.

4.1 Passive synchronization Design using InfiniBand Remote Atomics

In this section we discuss the issues and design challenges in implementing an efficient MPI-2 passive one-sided communications. Locks are used to protect accesses

to the protected target window affected by RMA calls issued between lock and unlock calls and to protect local load/store accesses to a locked local window executed between the lock and unlock call. MPI-2 passive synchronization supports locking in two modes: (i) exclusive mode and (ii) shared mode. Accesses that are protected by exclusive locks will not be concurrent at the window site with other accesses to the same window that are lock protected, i.e, only one process can have exclusive access to a window at a time. Shared lock mode allows multiple processes (readers/writers) to access the target window simultaneously. Accesses that are protected by a shared lock will not be concurrent at the window site with accesses protected by exclusive lock to the same window.

There are several different approaches for implementing passive synchronization. The passive synchronization could be implemented on top of two-sided communication. Another approach to implement passive synchronization when the memory window is not directly accessible by all the origin processes is by the use of an asynchronous agent at the target. This agent can cause progress to occur. One approach is to use a thread that periodically wakes up and checks for any pending one-sided requests. If there is underlying hardware support, then it can be exploited to provide truly one-sided passive synchronization.

There are several optimizations that are applicable to two-sided based approaches [60]. WMPI explored thread based one-sided communication and synchronization [44]. Previous work in MVAPICH2 used InfiniBand atomic operations to implement exclusive locks [37]. This design has some limitations that it considers only locking in the exclusive mode. In addition, this design does not guarantee immediate progress of the one-sided operations which are deferred to the unlock phase. This can hurt the

overlap potential. Our new design takes a step further and aims to address the above limitations while taking a similar approach of using hardware atomic operations. We use MVAPICH2 [46] as the framework for our design. In the current version of MVAPICH2, the passive synchronization support is based on two-sided communication primitives.

In this context we describe the two main aspects of our design: (i) efficient passive synchronization with support for locking in both exclusive and shared modes and (ii) enhancement of the scope for providing good overlap that the one-sided applications can potentially leverage. The following sections look at the design in further detail.

Efficient passive synchronization support can be designed using InfiniBand's remote atomic operations. Locking in exclusive mode can be implemented using InfiniBand's atomic compare and swap operations. This approach does not involve the remote process, and hence is a truly one-sided mechanism for passive synchronization. However, since MPI-2 allows for both shared and exclusive modes of locking for passive synchronization, it is imperative that our design allows for shared mode locking to co-exist as well. The current MVAPICH2 design provides two-sided based shared mode locking and this can be extended to work coherently with our design based on remote atomic operations for exclusive mode locking.

For every window on a target process we maintain a 64-bit global lock state that is registered with the NIC to support remote atomic operations. This 64-bit variable can be accessed using RDMA atomic operations. This global lock state variable can be in one of 3 states: (i) unlocked, (ii) locked in exclusive mode and (iii) locked in shared mode.

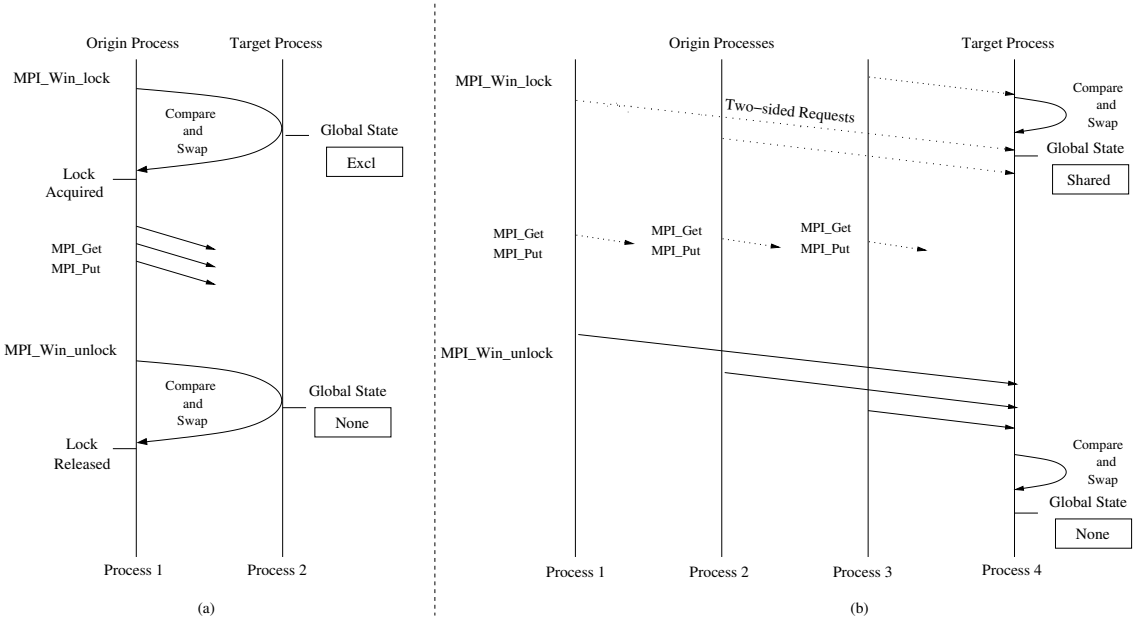

Figure 4.2: Locking Mechanisms:(a)Handling Exclusive Lock and (b)Handling Shared and Exclusive Lock

This variable is by default initialized to the unlocked state during window creation. *MPI_Comm_size* is used to indicate this unlocked state. To obtain an exclusive lock as seen in Figure 4.2(a), a network based atomic *compare-and-swap* operation is done on this variable. If the *compare-and-swap* is successful, then the lock is obtained and the global state variable is set to the *process rank* of the origin process indicating that it is the current holder of the lock. During the unlock operation, this value is set back to the default value. Other processes trying to obtain a lock at the same time would fail and would keep trying till they obtain the lock once the holding process relinquishes the lock.

In the case of a shared lock, we use the existing two-sided approach in which a message is sent to an agent on the remote node. The agent queues up the requests

and performs the issuing of lock and unlock operations locally. However, this can lead to conflicts with the exclusive mode locking and additional mechanisms are needed to handle this case.

To allow both shared and exclusive mode locking we use the following coordination mechanism. When a shared lock request is received by the remote agent it also performs an atomic *compare-and-swap* operation with the global lock state variable. If it can obtain the lock, that means there are no exclusive locks on this window, it sets the variable to a predefined value (*MPI_Comm_size + 1*) indicating that the lock is currently issued in shared mode and therefore all exclusive lock operations will be stalled. This agent also keeps a counter for the number of shared lock requests. When unlock operations are called, it decreases the counter variable. Once the counter variable reaches zero, it performs a *compare-and-swap* operation on global state variable resetting the global state value to the default *no lock* state. Figure 4.2(b) shows the basic protocol for shared mode locking. The dotted arrows in the figure indicates the operations that could be deferred to actually occur in the unlock phase when using the two-sided mechanism for obtaining shared locks.

The hardware based remote atomic operations have good scalability, but they might have the problem of flooding the network when the contention for locks is very high. However, mechanisms like exponential back off can be used to improve performance in such scenarios [37]. Since this is an orthogonal issue from the focus in this work, we have not incorporated this in our current design. We would like to incorporate this in the future.

4.2 Improve Overlap Scope for MPI-2 One-Sided Operations

Another aspect we aim to highlight by using the truly one-sided passive synchronization is to improve the overlap potential of the application. When two-sided approaches are used, the communication operations are often delayed to the synchronization phase and in some cases combined with an unlock synchronization call. In order to improve the progress, which leads to better overlap, we make sure that the one-sided operations within the passive epoch are issued immediately using the RDMA Write and RDMA Read InfiniBand operations. The completion of these operations are handled in the unlock operation. InfiniBand has limitations on the number of outstanding RDMA read and write operations. Hence to handle this in our design, additional requests beyond this limit are queued up internally and issued as soon as possible.

4.3 Overlap Analysis

In this section we analyze the different designs to understand the potential for overlap while using passive synchronization. In a passive synchronization mode overlap can be achieved at the sender side as well as the receiver side. In the sender side case, overlap can be more easily understood. Within the passive synchronization access epoch we could have computation and one-sided communication operations. If the one-sided routines are non blocking and can be initiated, then potentially we can perform computation while the initiated communication occurs in the background. More explicitly, we can do computation between MPI_Get or MPI_Put and the ensuing MPI_Win_unlock operation as long as this computation is independent or does not need the data from the one-sided operation. We refer to this as the *sender side*

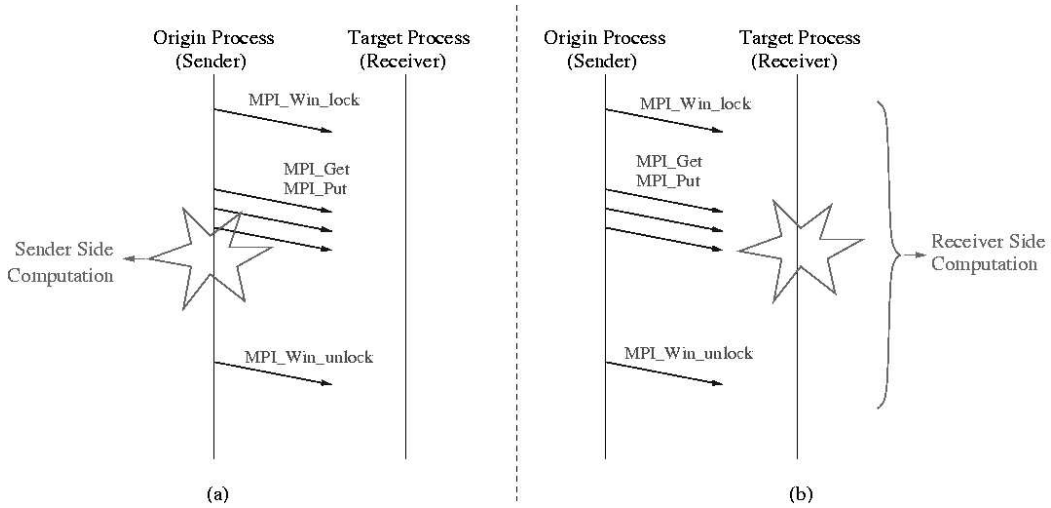

Figure 4.3: Computation and Communication Overlap: (a) Sender Side Overlap and (b) Receiver Side Overlap

overlap as shown in Figure 4.3(a). In addition to the sender side overlap in a passive synchronization mode we can have computation on the target node while communications are occurring in its target window. This could be thought of as *receiver side overlap* as seen in Figure 4.3(b). An MPI-2 one-sided library geared towards maximizing overlap should provide both these kinds of overlap benefits to the application to the extent possible.

In this context, we try to analyze the two described approaches from the overlap perspective. In the current two-sided approach there is a remote agent or receiver (in the MPI library) that handles all the one-sided communication/synchronization requests including lock, unlock, get, put. On the sender side (origin process) the lock is a local operation that is queued. The data transfers are also queued and it is only in the unlock phase that the entire lock/data transfer and unlock occurs. This kind

of implementation is good when there is a requirement for lower overhead synchronization operations. Further, in this case the data transfer and the synchronization messages can be combined thus reducing the number of required network operations leading to benefits in certain scenarios. However, this results in extremely poor overlap capability for an application. Though lowering the overhead or latency of the synchronization is important, it should not come at the cost of reducing the overlap potential. Since the data transfer occurs in the unlock phase, any computation in the passive epoch cannot be overlapped at all. Also the two-sided approach requires the target node to be involved in both the computation as well as the synchronization calls. Hence this affects the on-going computation on the target node thus resulting in lower receiver overlap too. Whereas in the *direct passive* approach, the synchronizations as well as the communication operations are issued as early as possible. Further all these operations are truly one-sided because they use the underlying RDMA operations. Hence we expect better computation and communication overlap on both the sender and receiver side for the *direct passive* approach.

4.4 Performance Evaluation

In this section we present the experimental evaluation of our *direct passive* implementation. We analyze the overlap scope with the two-sided based and *direct passive* implementations. It is to be noted that we use locks in exclusive mode for our evaluation. We then describe the results for the modified MPI-2 version of the SPLASH LU benchmark [54]. This version was obtained by modifying a shmem version of SPLASH LU benchmark to use MPI-2 one-sided calls with passive synchronization.

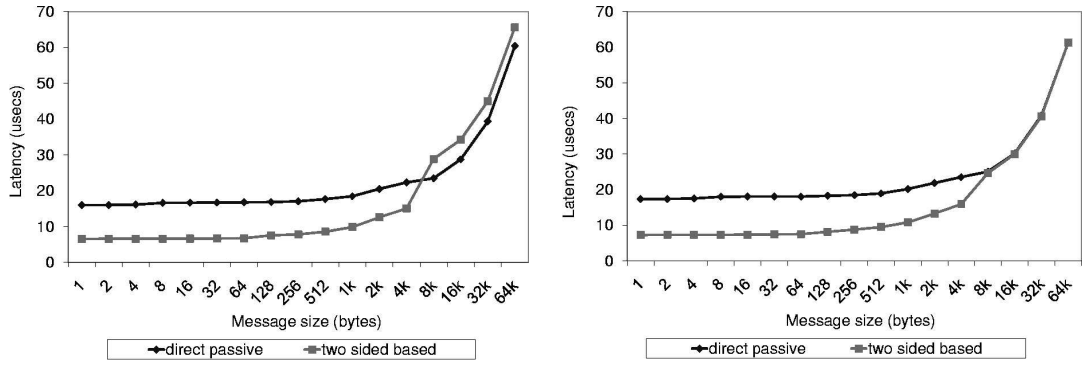

Figure 4.4: Basic Passive Performance of (a) Put and (b) Get operations

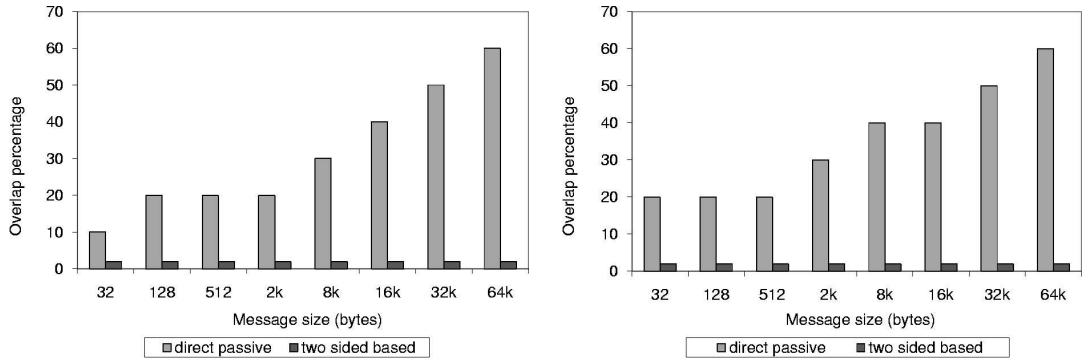

Figure 4.5: Overlap Benefits of Basic One-sided operations: (a) Put and (b) Get

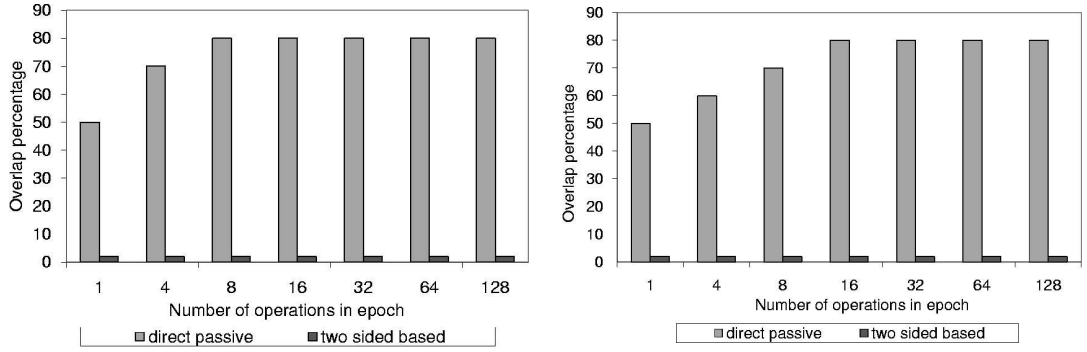

Figure 4.6: Overlap Benefits with Increasing Number of Operations: (a) Put and (b) Get

We further profile the results of this SPLASH benchmark to analyze the performance in greater detail.

Our experimental testbed is a 64 node Intel cluster. Each node of our testbed is a dual processor (2.33 GHz quad-core) system with 4 GB main memory. The CPUs support the EM64T technology and run in 64 bit mode. The nodes support 8x PCI Express interfaces and are equipped with MT25208 HCAs with PCI Express interfaces. A Silverstorm 144 port switch is used to connect all the nodes. The operating system used is RedHat Linux AS4.

4.4.1 Microbenchmarks

In this section we compare our new passive design with the existing design using microbenchmarks that measure latency and overlap capabilities.

Overall Latency using Passive Synchronization

First we compare the basic performance of the two approaches: not just the cost of synchronization, but from the perspective of data communication using passive synchronization. This is often more representative of application behavior. We measure the time taken or latency for a lock operation followed by put and an unlock operation for various message sizes. This benchmark shows the overall latency of the two approaches.

The results are shown in Figure 4.4. As seen in the figure, the two-sided approach performs better than the *direct passive* scheme for small messages. This is because for small messages the synchronization overhead is a significant ratio of the total time. i.e. the *direct passive* scheme needs two RDMA atomic compare and swap operations for synchronization in addition to RDMA read/write communication operation. The

overhead is lower in the two-sided approach since it can combine the communication and synchronization in a single message. For larger messages, the *direct passive* scheme performs better or equally well, as the cost of data transfer is dominant. However as we have discussed in earlier sections, latency alone is not the main metric. The amount of the overlap capability the implementation can provide is critical to the performance of a one-sided application. Hence we study the designs from the overlap perspective in depth in the following section.

Overlap Potential

In this section we come up with a set of micro-benchmarks that can evaluate the overlap potential both at the origin as well as the target process.

Sender Side Overlap: In this benchmark we evaluate the sender side overlap. The following is a brief description of the benchmark. Process 0 (origin process) does a lock/put/unlock on the window located on the remote target process. The test estimates the time for the lock/put/unlock sequence. Between the get call and unlock synchronization call, increasing amounts of computation as a percentage of the estimated time are introduced. As long as the overall execution time does not change, it implies that the computation time is being absorbed or overlapped with the issued communication call. The results for this are shown in Figure 4.5(a). The *direct passive* implementation shows very good overlap for large messages whereas in the two-sided approach virtually no overlap is possible because all the data transfer operations occur in the unlock phase. Please note that for the sake of visibility in the graph, we have shown a small value for the two-sided approach which can essentially be ignored. Similar results are seen for lock/get/unlock sequence shown in Figure 4.5(b).

Sender Side Overlap with Varying #operations in Epoch: This benchmark is an extension of the previous benchmark where we vary the number of get/put calls between the lock and unlock operations. The message size used is 32K. This test tries to mimic application scenarios where multiple get and put calls are issued between the synchronization operations in order to amortize the overhead of synchronization. As in the previous test increasing amounts of computation is introduced. Figure 4.6 shows the results of this benchmark. Once again the *direct passive* approach is able to provide much higher overlap as opposed to *no-overlap* for the two-sided approach.

Receiver Side Overlap: This benchmark tries to measure the impact of target involvement in passive mode communication on the ongoing computation. In this test there is one origin process and one target process. The test performs a fixed amount of computation on the target node. The execution time of this benchmark is the time taken by the target node to perform the fixed amount of computation. At the same time the origin process tries to access the memory window using MPI_Get operations within a lock/unlock passive epoch. This test in effect tries to measure receiver (target) overlap, i.e, it tries to measure how much of the computation on the target node can be overlapped with the ongoing communication operations.Figure 4.7(a) shows the normalized execution time of this benchmark. As compared to execution time with the *direct passive* scheme (which is normalized to 1), we observe that the two-sided approach leads to considerably higher execution times. This indicates the overhead of the target involvement for the two-sided approach or in other words this shows the reduced overlap (or lack thereof) on the target node.

Receiver Side Overlap with Multiple Origin Processes: We further extend the receiver overlap benchmark to multiple processes emulating one-sided application

patterns. In this benchmark the overlap capability is observed in the presence of increased accesses to the target window. The message size used is 32K. Figure 4.7(b) shows the results in terms of normalized execution time. We see that for 64 processes, the deterioration in the execution time is about 4.5 times worse for the two-sided case as compared to *direct passive*. This is largely because of the increased communication overheads on the target node for the two-sided approach which delays the computation adversely affecting the overall execution time.

4.4.2 Application evaluation with SPLASH LU benchmark

In this section we use a modified version of the SPLASH LU benchmark to demonstrate the benefits of overlap for an one-sided application. The SPLASH LU benchmark does dense LU factorization. The dense n x n matrix is divided into an N x N array of B x B blocks, such that n=NB. The blocks of the matrix are assigned to processors using a 2D scatter decomposition. The communication in LU occurs when a diagonal block is used by all the processors that require it to update the perimeter blocks they own and when the perimeter blocks are used by all processors that require them to update their interior blocks. We modified a shmem version of SPLASH LU benchmark to use MPI-2 one-sided operations. We use MPI_Get calls to fetch the block of data and we use MPI_Win_Lock/MPI_Win_unlock passive synchronization calls. The MPI_Win_lock calls are used in exclusive mode. The problem size gives the size of the overall matrix and we can vary the block size. We show the results for this benchmark for varying problem sizes and a block size of 128. This block size gave the best results.

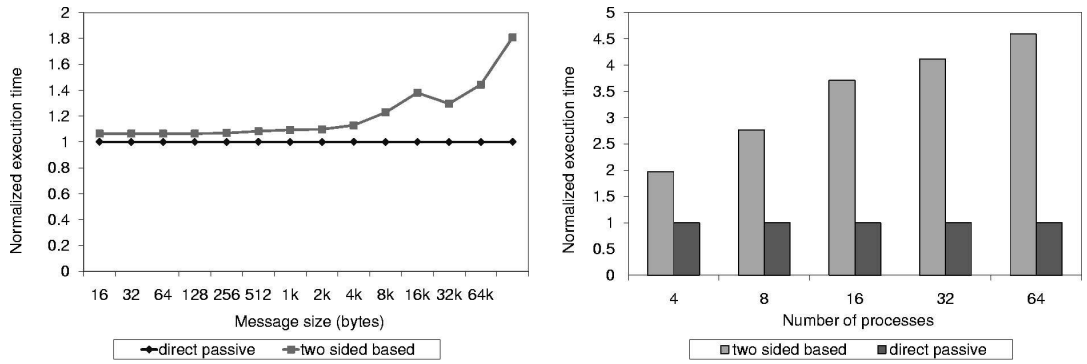

Figure 4.7: Receiver overlap capability with (a) two process and (b) multiple processes

Figure 4.8 shows the performance of the MPI-2 SPLASH LU benchmark for the two approaches. We observe that the *direct passive* approach always outperforms the two-sided approach. This is because the *direct passive* approach provides better overlap with reduced remote CPU involvement. In Figure 4.8(a) we show the performance of SPLASH LU with a problem size 2048. We observe that the *direct passive* approach performs about 25% - 81% better than the two-sided approach. Figure 4.8(b) shows the performance for a larger problem size of 3000. In this case we observe higher gain ranging from 58% - 87% for the *direct passive* case as compared to the two-sided case.

In order to further understand these results, we profile the application run. In this we measure the average time spent by the application in each of the MPI library calls. In particular, the only relevant MPI calls used in the SPLASH LU code are MPI_Win_lock, MPI_Win_unlock, MPI_Get and MPI_Barrier. The remaining time is classified as computation time. In Figure 4.9(a) we show the timing break up of these operations for problem size 2048 for 8-64 processes. The results for problem size 3000 are shown in Figure 4.9(b). The legends with T stand for the two-sided based approach, and with O stand for the one-sided *direct passive* approach. As

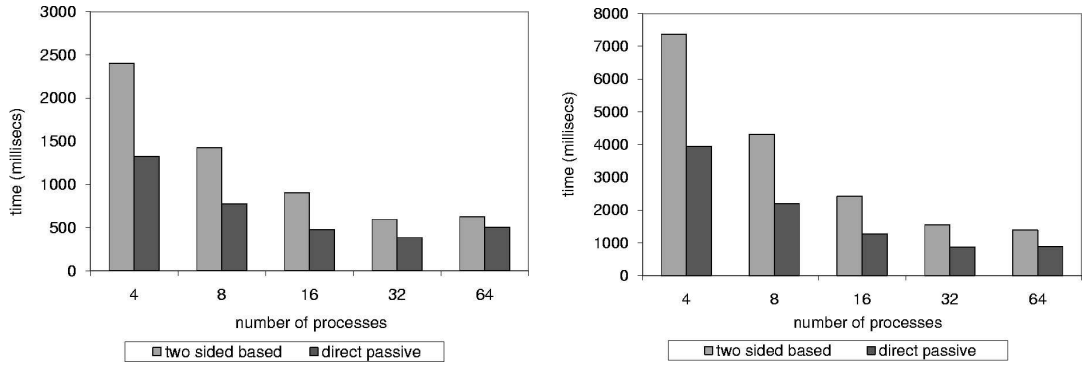

Figure 4.8: MPI-2 SPLASH LU benchmark: (a) Problem Size 2048 and (b) Problem Size 3000

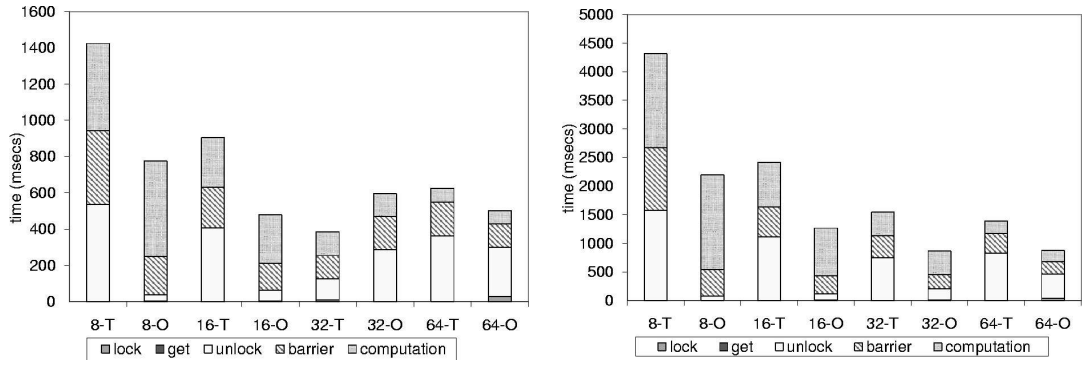

Figure 4.9: Timing Breakup of MPI-2 SPLASH LU: (a) Problem Size 2048 and (b) Problem Size 3000

discussed in Section 4.3 for the two-sided approach, we observe that the lock and get operations for the two-sided approach take negligible time, since these operations are queued locally. The actual progress of these operations occurs in the unlock phase, i.e, the operations are initiated during the unlock operation. We see that the unlock operations in this case take a large amount of time as expected. On the target node, the progress for these operations is delayed and triggered only during the MPI barrier calls. This is due to the fact that passive synchronization do not have explicit progress calls for the target node.

On the other hand, the *direct passive* scheme acquires the lock and initiates the one-sided RDMA data transfers immediately and the progress of these operations are transparent to the target node. Since this does not need the remote process to intervene, the remote process makes faster progress on its own tasks. In addition, since the MPI Get operations do not need to wait for the target node to trigger progress, these operations move ahead faster reducing the overall application time. This aspect is clear from the numbers in Figure 4.9 where the two-sided approach spends a much larger time in the MPI Barrier time in performing the remote get requests which delays the computation. Consequently we also observe that the unlock time taken for the two-sided cases is significantly higher (832ms for problem size 3000 and 64 processes) as compared to the *direct passive* (421ms for problem size 3000 and 64 processes).

To improve the performance of MPI-2 one-sided communication, in this work, we focussed on the following important aspects: (i) *direct passive* synchronization support using InfiniBand atomic operations and (ii) enhancement of one-sided communication progress to provide scope for better overlap that one-sided applications

can leverage. In addition we performed an in-depth study to characterize the sender side and receiver sideoverlap capabilities of our *direct passive* design.

Our evaluation shows significant improvement in the overlap potential for the *direct passive* design that can be leveraged by a one-sided application. Our micro-benchmarks show that the overlap on both the sender and receiver side is significantly enhanced using our approaches. In addition to the micro-benchmarks we also demonstrate a significant improvement ranging between 58% - 87% in the performance of an MPI-2 one-sided version of the SPLASH LU benchmark as compared to the existing design. Our detailed analysis shows that the potential benefits in this case come from the reduced remote side involvement that is achievable by our design.

4.5 Related Work

There are several studies regarding implementing one-sided communication in MPI-2. Some of the MPI-2 implementations that support one-sided communication are: MPICH2 [9], WMPI [44], NEC [63] and SUN-MPI [16]. Besides MPI, there are other programming models that use one-sided communication. ARMCI [47], GASNET [15] and BSP [28] are some examples of this model.

Researchers in [21] have proposed distributed queue based DLM using RDMA operations. Though this work exploits the benefits of RDMA operations for locking services, their design can only support exclusive mode locking. Further, prior research in [45] extensively utilizes InfiniBand's remote atomic operations for shared and exclusive mode locking, however, the main focus in their work is not in the context of MPI-2 one-sided synchronization but rather as a system-wide distributed locking

service typically used in data-centers. In the context of MPI, previous work in MVAPICH2 have studied the benefits of RDMA atomic operations to efficiently implement locks in exclusive mode [37]. However their design does not take shared locks into account. OpenMPI [12] is another open source MPI implementation that supports MPI-2 standards. In OpenMPI, the library is single threaded by default and uses the two-sided approach for passive synchronization currently and depends on the target process making MPI calls to make progress. Our new design goes a step further to address the limitations of these approaches. It provides exclusive lock mode using atomic operations and shared mode locking support by extending the existing two-sided based shared locking in the MPI library and also tries to maximize the overlap potential.

CHAPTER 5

MIGRATING LOCKS FOR MULTI-CORES AND HIGH-SPEED NETWORKS

Most processor architectures provide fast atomic locks based on few CPU instructions. These can be used to implement locks efficiently across processes within the same node. As described in previous chapter, networks such as IB provide network atomic operations that can be used to implement locks across nodes in an efficient and truly one-sided fashion. However, these two forms of locks are not interoperable. Specifically, network-based atomic operations achieve their atomicity through serialization at the network adapter. That is, the network adapter orders accesses to the atomic variable in the order in which it receives requests, thus guaranteeing that the variable is always in a consistent state. CPU-based atomic operations, on the other hand, do not pass through the network adapter at all, and are handled fully in the processor cache.

If both the CPU and the network try to work on the same lock, it is possible that the CPU fetches the variable to cache to perform an operation on it. At the same time, the network can trigger a cache flush through the chipset, forcing the variable to be in an inconsistent state.

In short, the CPU and the network need to work on different locks leading to several challenges in achieving lock coherence in a one-sided manner, that we will address in this work.

While using IB network atomic operations for one-sided communication allows for truly one-sided passive synchronization, this approach might not be the best in light of the increasing number of multi-core systems and the number of cores on each system. Specifically, using network operations to synchronize even between processes on the same node can have performance implications (since all the data has to traverse down to the network adapter and back) as well as network contention issues (since the network adapter is shared between all the cores). Thus, in this chapter, we propose a new hybrid migrating locks design shown in highlighted part of Figure 5.1 of our proposed research framework that utilizes CPU-based atomic operations in conjunction with network atomic operations to take advantage of both.

5.1 Proposed Hybrid Design

Simultaneously utilizing both CPU-based atomic operations as well as network atomic operations is not trivial because of interoperability issues between these two operations as discussed above. Thus, there has to be a coordination mechanism between the network based locks and the CPU based locks. Our proposed solution to the problem is to migrate between the two locking mechanisms (network locks and CPU locks) when required. Since the locking is per-window based, different windows on the same process could be in a different locking mode depending upon the nature of the lock requests for that window.

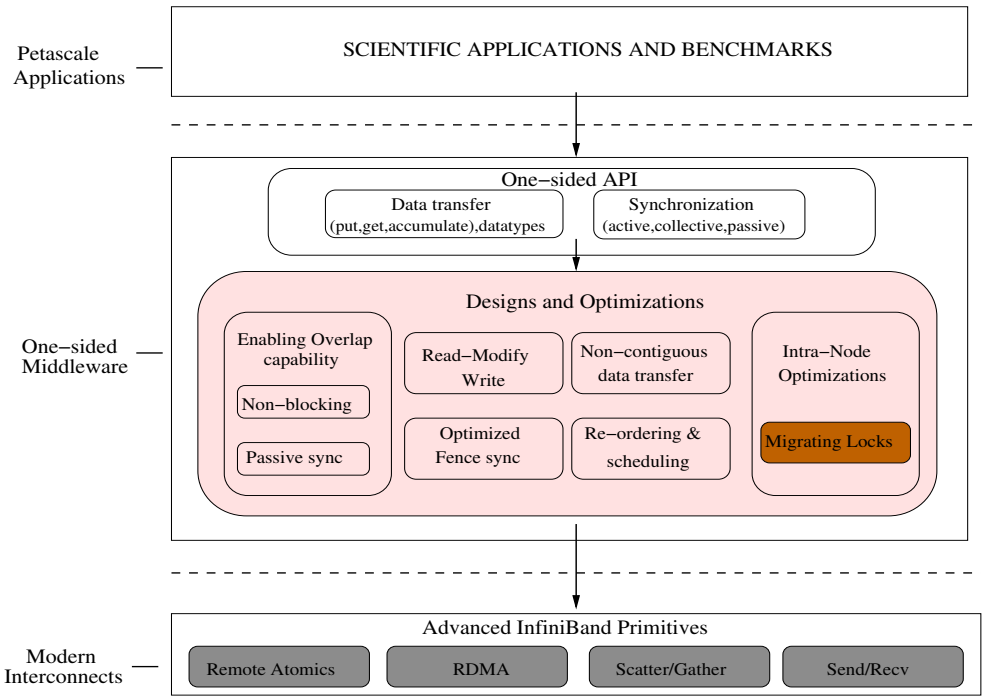

Figure 5.1: Overview

Every node maintains the following state variables: (i) locking mode (network or CPU based), (ii) CPU lock and (iii) 64 bit global network lock. The locking mode variable and CPU lock variable are placed in shared memory so that other processes on that node can access it. The network lock can have the following values: (i) a value of 0 to (*MPI_Comm_size* - 1) indicates that the lock is in network mode and the actual value denotes the process that holds the network lock, (ii) a value of *MPI_Comm_size* indicates that it is unlocked, and (iii) a value of *MPI_Comm_size* + 1 indicates that the lock is in CPU mode.

In the network lock mode described in Figure 5.2, all the locks use IB atomic operations to obtain the network lock. In the CPU lock mode described in Figure 5.3, the intra-node locks use fast CPU based locks and the inter-node locks use a two-sided

approach of sending the lock request to the lock manager (step 1) which then obtains the CPU lock on its behalf (step 2) and responds with lock granted (step 3).

By default, the lock is preset to one of the above two-modes, for example CPU based mode. When the mode needs to be migrated, a two-sided message is sent to the lock manager which acquires both the network as well as CPU lock, modifies the locking mode to 'network', and then grants the lock. Any further locking now happens through IB atomic operations in a completely one-sided manner. The lock migration from a CPU mode to network mode is illustrated in Figure 5.4. When a remote process wants to acquire a lock, it performs a compare and swap with the network lock state (step 1). If the remote process discovers that the lock is in CPU mode, and it wants to migrate the lock to network mode, it sends a two-sided message to the lock manager requesting migration to network mode (step 2). The lock manager acquires both the network lock and the CPU lock (step 3), modifies the lock mechanism to CPU mode (step 4), and sends the lock granted packet to the remote process (step 5). A similar approach is done to reset the lock to CPU based. In this way, the locks can be migrated from one mechanism to other.

Thus, in summary, intra-node locks are completely one-sided as long as the lock is in CPU-mode and inter-node locks are completely one-sided as long as the lock is in network mode. If the lock is not in the appropriate mode, a two-sided synchronization is needed to migrate the lock to the appropriate mode. Henceforth we will refer to this approach as 'Hybrid'.

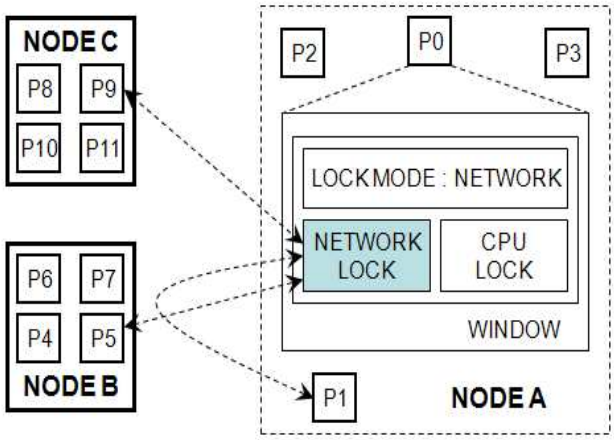

Figure 5.2: Locking Mechanisms: Network Lock

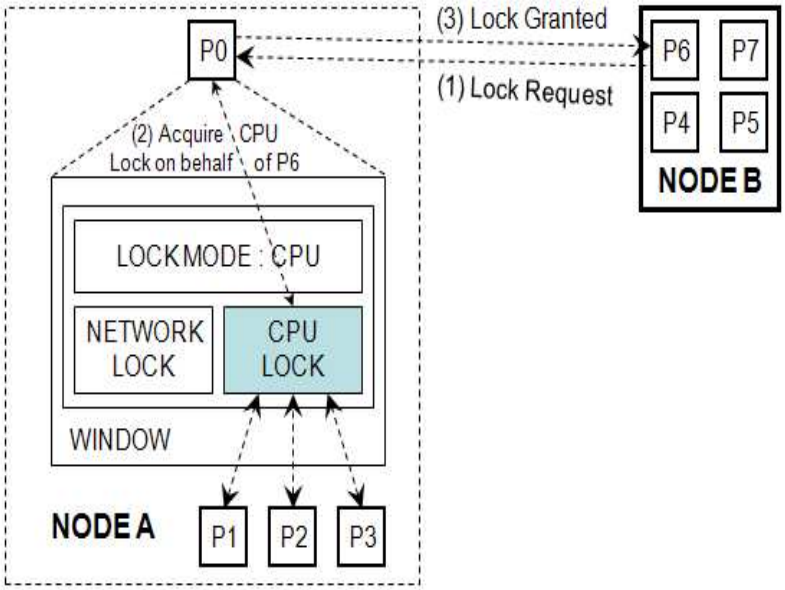

Figure 5.3: Locking Mechanisms: CPU Lock

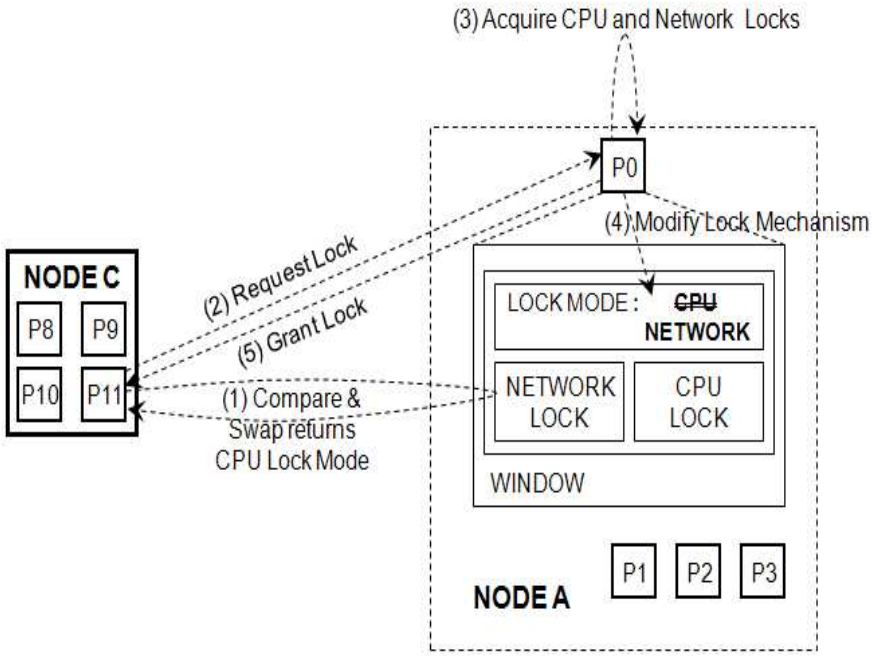

Figure 5.4: Locking Mechanisms: Lock Migration

5.2 Migration Policies

Migration of locks could be based on various criteria. It could be based on: (i) communication pattern, (ii) history, (iii) priority, (iv) native hardware capabilities and so on. The criteria used to migrate the locks is not the focus in this thesis, and could be part of follow up work. In all the evaluations in this paper, the lock is preset to CPU mode for simplicity. Any remote node process lock request migrates the lock to network mode and any future intra-node lock request migrates the lock to CPU mode.

5.3 Experimental Results and Analysis

In this section we evaluate the performance of our migrating locks based 'hybrid' design with the purely 'two-sided' based and the network based 'one-sided' approaches described in Section 4.1. We evaluate the performance for a wide range of scenarios. First, we evaluate and analyze the performance when the lock/unlock operations occur within the same node (intra-node) among the different cores. Then we show the performance when the operations are purely inter-node. Then, we evaluate the performance for a combination of inter-node and intra-node operations. We also measure the overhead involved when the locks are migrated. Finally we evaluate the performance for SPLASH LU benchmark.

Experimental Testbed

Each node of our testbed has 16 AMD Opteron 1.95 GHz processors with 512 KB L2 cache. Each node also has 16 Gigabyte memory and PCI-Express bus. They are equipped with MT25418 HCAs with PCI-Ex interfaces. A 24-port Mellanox switch is used to connect all the nodes. The operating system used is RedHat Enterprise Linux Server 5.

5.3.1 Intra-node Performance

In this section, we first evaluate the performance of our new design for intra-node operations on a single node. Figure 5.5 shows the performance of lock/unlock operation comparing the three approaches. As expected our new hybrid design performs the best, since the lock/unlock operations within a node are basically few CPU instructions. In the two-sided approach, a lock request packet is sent to the lock

manager of the target process. The lock manager responds with the lock granted packet. These lock requests and lock granted packets go over shared memory since the target is on the same node. In the one-sided based approach, the lock operation is achieved through an IB loop-back atomic fetch and add operation. Since the loop-back operation is expensive, it has the lowest performance for a single lock/unlock operation.

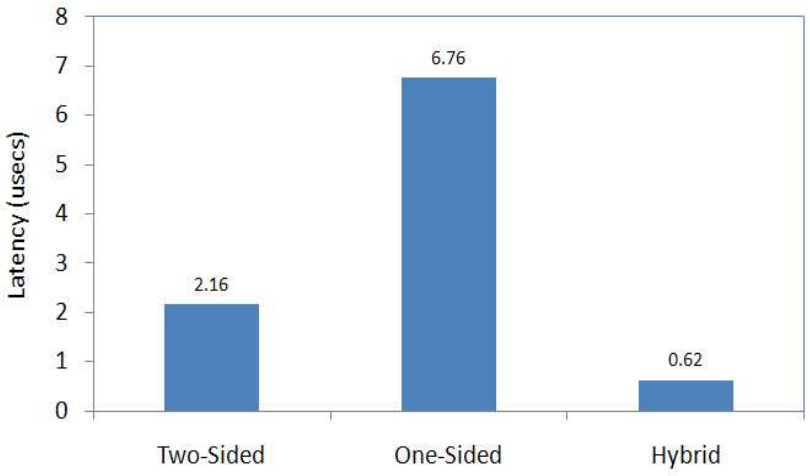

Figure 5.5: Lock/Unlock Performance

Intra-node Performance with Remote Computation

Next we evaluate the performance of the three approaches in the presence of computation on remote/target process. Minimal remote/target process involvement is important for one-sided passive synchronization calls so that the target can proceed with its computation. In this benchmark, the origin process acquires the lock and unlock operation on target process while computation is performed on the target

process. The computation is a dummy loop that is executed on the remote/target process. In this experiment the performance of the three schemes is measured for varied amounts of dummy loop computation. The results are shown in Figure 5.6. Here the one-sided approaches (network based, one-sided and hybrid approach) are not affected with increasing amounts of computation on the target process, since they are not dependent on the target process to progress. Whereas, the performance of the two-sided scheme degrades with increasing amount of computation. This is expected because the two-sided approach requires target process involvement. In the presence of computation, it takes longer to respond to the lock/unlock requests.

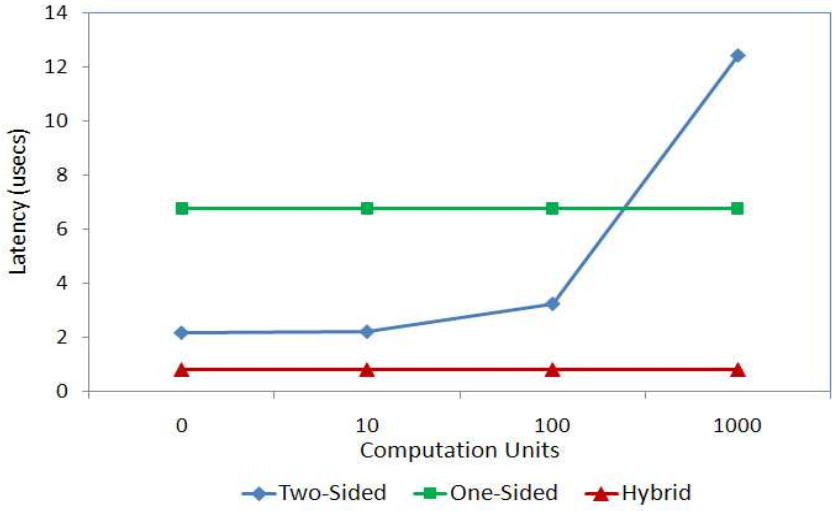

Figure 5.6: Lock/Unlock Performance with Remote Computation

5.3.2 Concurrency and Contention

Next we evaluate the performance of the different approaches when several lock/unlock operations occur concurrently. These experiments are conducted on a single node.

Network Contention

In the first micro-benchmark, each process locks its neighboring process (rank+1) on the same node. Thus in this benchmark, there are as many lock/unlock operations happening concurrently as the number of cores for which the benchmark is run. We measure the average latency of lock/unlock operation in this scenario. The results are shown in Figure 5.7. We observe that the two-sided performance is not degraded since the lock/unlock requests messages are sent over shared memory and there is no network contention. However the one-sided scheme using loop-back suffers degradation due to network contention since all the lock/unlock operations result in network transactions. In this scenario also, the hybrid scheme performs the best since the CPU based locks do not result in network contention.

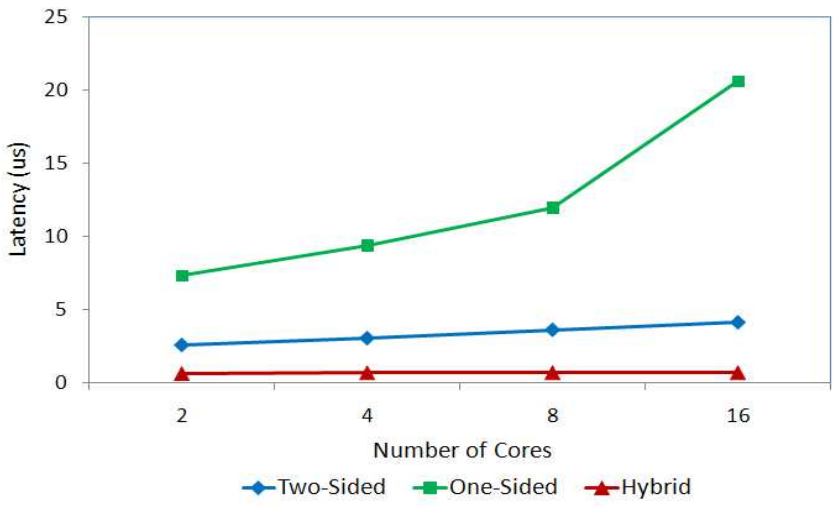

Figure 5.7: Lock/Unlock Performance with Network Contention

Lock Contention

The next benchmark shows the performance of the three approaches when several processes are contending for a lock on the same window. The results are shown in Fig. 5.8. The hybrid scheme performs the best for up to three lock contentions. Beyond four contentions, the two-sided approach performs better than the hybrid scheme. The one-sided approach performs the least. This is expected since there would be lots of network transactions in the presence of contention.

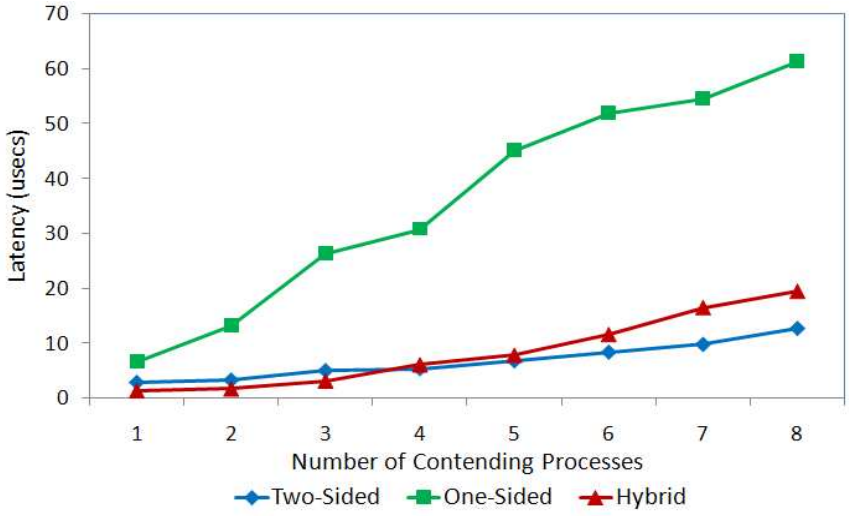

Figure 5.8: Lock/Unlock Performance with Lock Contention

5.3.3 Inter-node Performance

In this section, we compare the performance of the three approaches when the operations are purely inter-node. We use a micro-benchmark to demonstrate the benefits of one-sided approaches in the presence of computation and skew. We used Testbed B for this experiment, since we had more number of nodes to understand

the inter-node performance. The experimental testbed (Testbed B) used for this benchmark is a 64 node Intel cluster. Each node of the testbed is a dual processor (2.33 GHz quad-core) system with 4GB main memory.

The benchmark simulates a ring type of communication wherein each process locks the window of its successor, puts some data in the target window and updates a tag indicating completion of the data transfer to that window. The target process then makes sure that the data is available in its window, then performs the same operation on its successor. The communication terminates when the message traverses through the complete ring. Simultaneously all the nodes are also performing computation in the form of a dummy loop. For the sake of simplicity, a fixed amount of computation is being performed by all the nodes. This benchmark evaluates the capability to overlap computation and communication. The results are shown in Figure 5.9. The one-sided and the hybrid approach outperforms the two-sided approach. This is due to the ability of the one-sided and hybrid approach to perform the lock/unlock operations in a truly one-sided fashion, whereas the two-sided approach requires remote host involvement to make progress. This results in delay for the target process in responding to lock requests. Since this benchmark is a ring type of communication, this could manifest itself as skew for the other processes further in the ring resulting in a cascading effect. In this scenario, the hybrid scheme remains in the network locking mode exclusively and hence its performance is similar to that of the one-sided approach.

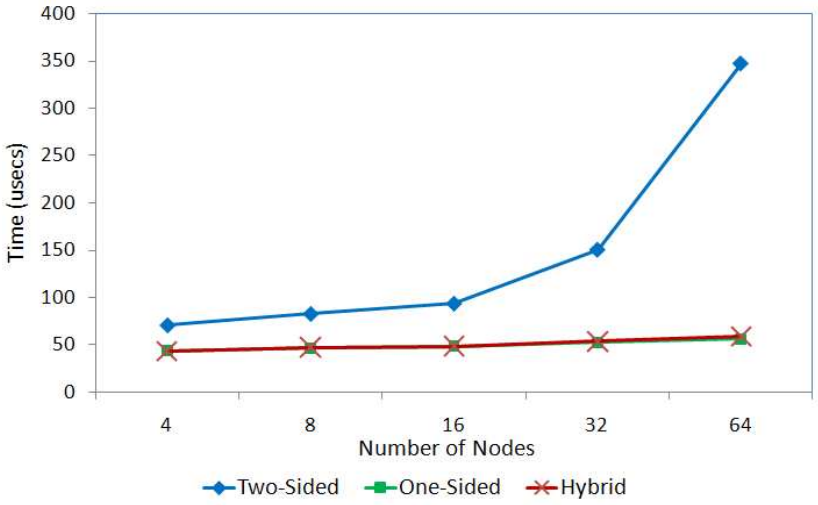

Figure 5.9: Inter-node Performance

5.3.4 Lock Migration

In this section, we try to evaluate the overhead incurred due to lock migration. The benchmark measures the average time taken for an intra-node lock/unlock operation and an inter-node lock/unlock operation in the presence of migration of the lock mechanism from network mode to CPU mode and vice-versa. The experiment is a two node experiment in which a process P1 acquires a lock/unlock on a process P0 on the same node 1000 times. During this duration, a process P2 on the second node tries to obtain the lock on P0 for x times triggering a migration each time.

The intra-node line in Figure 5.10 shows the latency of the lock/unlock operation happening on the same node with increasing percentage of migrations. We observe that for small percentage of migrations, the overhead is not very high as compared to case when no migrations occur. The inter-node line similarly shows the latency of the lock/unlock operation happening across nodes with increasing percentage of

migrations. For smaller number of migrations, the overhead incurred is quite less. Large number of migrations lead to some overhead. However it is to be noted that, the biggest benefit achieved by this approach is to be able to maintain the truly one-sided nature of the locks once the migration has been achieved and thus provide greater potential for asynchronous communication as well as higher computation communication overlap. Also the migration policy described in Section 5.2 can be used appropriately to minimize the number of migrations.

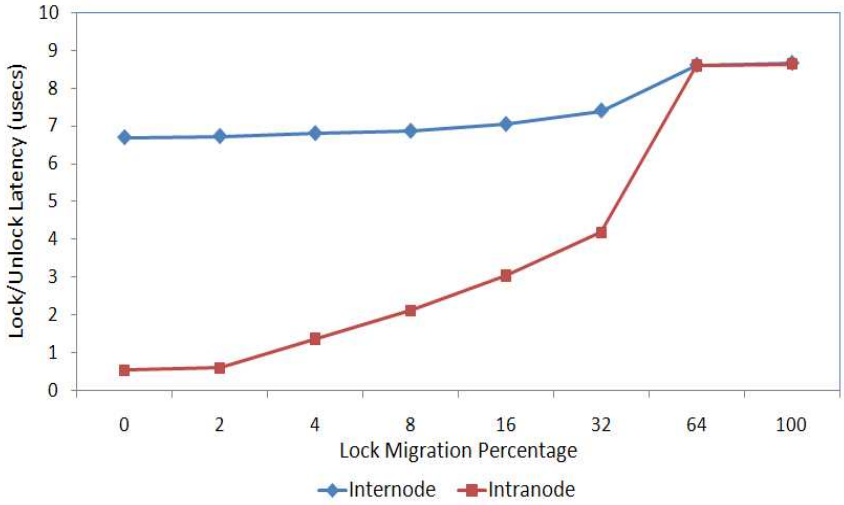

Figure 5.10: Lock Migration Overhead

5.3.5 Hierarchical Task Sharing Communication Pattern Micro-benchmark

In this section, we evaluate the performance for a combination of inter-node and intra-node operations with lock migrations by simulating a benchmark that performs task sharing and redistribution. The details of the benchmark is described below.

The experiment is run on 4 nodes with 16 cores on each node for a maximum total of 64 cores. A hierarchy of leaders is created with one leader process designated on each node. First, the leader on every node performs 1000 Lock-Put-Unlock on every other local process on the same node. Then, the leader performs 1000 Lock-Put-Unlock on the leader of every other node. Finally, the leader on every node performs 1000 Lock-Put-Unlock on every local process again. The benchmark tries to simulate a scenario in which a leader process tries to get data/work from close neighbors, then gets data from remote neighbors in a cycle. The resulting communication pattern is a clique-based communication described in earlier sections. The results are shown in Figure 5.11. The communication pattern described above has lot more intra-node operations than inter-node operations. The hybrid scheme performs the best because it uses the fast CPU locks for the intra-node operations, and when the operations are inter node, it migrates to network mode. Thus it provides the best performance for such a communication scenario and we also observe that the performance gap is sustained for increasing number of processes.

5.3.6 Evaluation with SPLASH LU benchmark

In this section we evaluate the performance of the the three schemes using a modified version of SPLASH LU benchmark. The SPLASH LU benchmark was modified to use MPI-2 one-sided communication. It uses MPI_Win_lock/MPI_Win_unlock passive synchronization operations and uses MPI_Get operations to fetch the block of data. The MPI_Win_lock calls are used in exclusive mode.

The results are shown in Figure 5.12. The x axis gives the number of processes (a*b indicates a - number of nodes, b - number of cores per node) and y axis shows

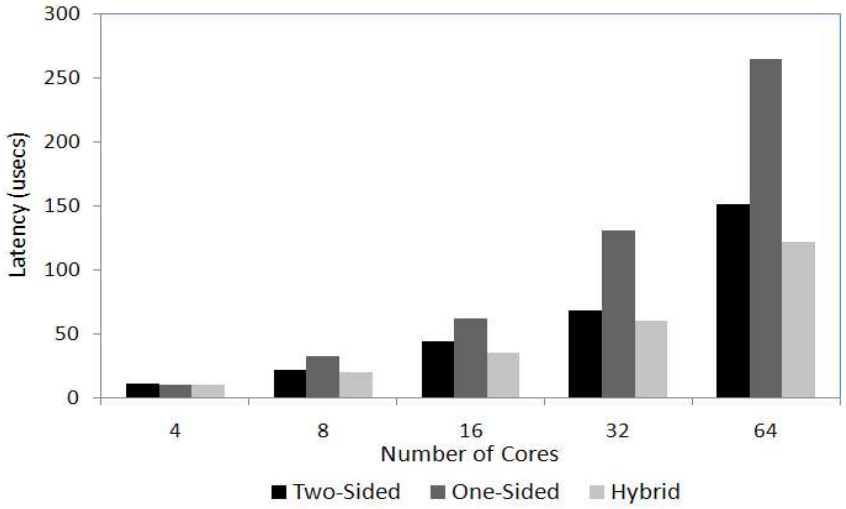

Figure 5.11: Hierarchical Task Sharing Communication Pattern

the time taken in milli seconds for problem size 2048. Here we observe that the hybrid scheme performs the best when all the processes run on one node. For all the other cases the two-sided approach performs the best and the hybrid scheme fares badly. To understand this better, we profiled the number of inter-node and intra-node operations as well as the number of migrations occurring for the hybrid approach during the benchmark run. These results are shown in Table 5.1 and Table 5.2.

For the one node case, all the operations are intra-node operations. In this case the hybrid scheme uses the fast CPU locks and there is no migration at all during the run. Hence in this case the hybrid approach gives the best performance. The one-sided case performs the worst as expected. For the other cases (2*8, 4*8 and 8*8), we have both inter-node and intra-node operations. Also with more nodes, the percentage of inter-node operations in the SPLASH LU benchmark become more significant, around 90% in case of 8*8 configuration. At the same time we also observe that the total

number of lock migrations for the hybrid scheme increase with increasing number of nodes.

The poor performance of the one-sided design could be attributed to the overhead of the loop-back operations for intra-node operations as well as network contention. For the hybrid approach, the number of migrations seems to significantly affect the performance of the hybrid design. During migration, both the network lock and the CPU lock needs to be acquired before the mode can be switched. If several local lock requests occur concurrently, it is possible that it takes a longer time to acquire both the network and CPU locks in order to modify the lock mode. This could result in poor performance. In such situations it would be better for the lock manager to keep track of the incoming lock request pattern and yield the lock. The current design does not keep track of such information. Also the existing migration policy leads to frequent migrations.

One possible enhancement is for the migration policies to take into account the arrival pattern of the lock requests and grant the requests more intelligently.

Numprocs	Intra-Node Locks	Inter-Node Locks
1*8	57744	0
2*8	36144	59840
4*8	14560	114704
8*8	14560	192080

Table 5.1: Inter-node vs Intra-node locks

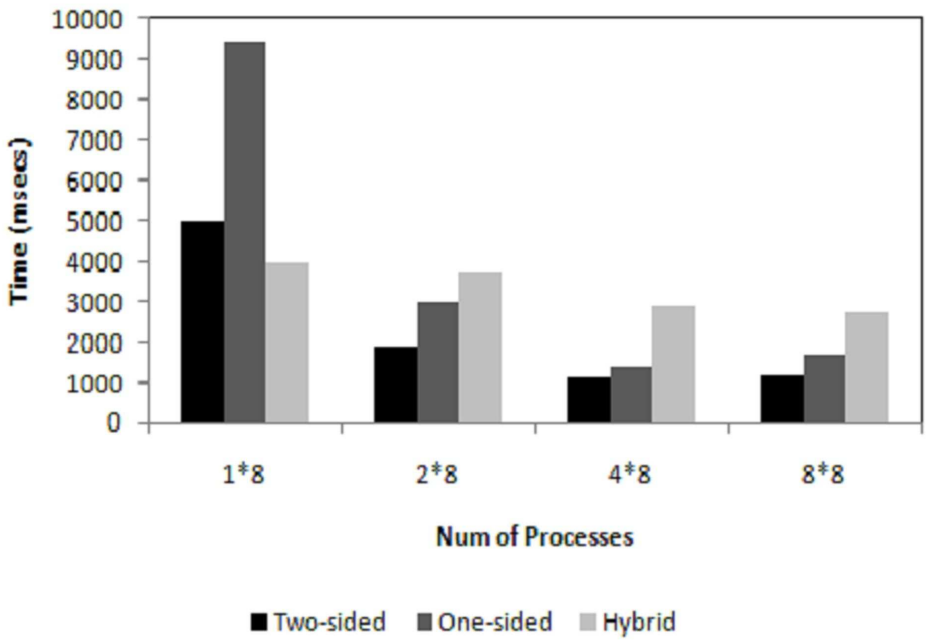

Figure 5.12: SPLASH LU Benchmark

5.3.7 Discussion

As seen from the above results, the performance of the hybrid approach is dependent on the pattern of the communication operations. Based on the results from the SPLASH benchmark result, a naive migration policy of migrating for every request is not a good choice. Further the lock manager needs to be enhanced to keep track of the state of the different incoming requests as well as the history of incoming requests so that it can make decisions more intelligently. Also different migration policies need to be implemented and evaluated.

Another aspect that is important is how can the application writers/users take advantage of the migration policies. If the users are aware that there are going to be very few inter-node operations, or in the case where the hardware does not support

Numprocs	Migrations
1*8	0
2*8	9346
4*8	10180
8*8	16400

Table 5.2: Num of Migrations

network locks, then the lock mechanism can always be set to the CPU mode and the inter-node locks can use two-sided based approach. The user can also specify to the library that the lock should be switched only after a certain number of network lock / CPU lock requests occur back to back so that the lock migrations do not occur frequently. Another approach is to pass communication pattern information as well as other guideline information to the MPI library in the form of hints. MPI standard supports the interface for providing hints to the library. This can be used to give priority to a particular lock operation for instance.

5.4 Related Work

There are several studies regarding implementing one-sided communication in MPI-2. Most of the related work has been described in Section 4.5 of the previous chapter.

Further researchers in [17] have studied efficient implementation of locks using NIC based atomic operations on Myrinet.

CHAPTER 6

FENCE SYNCHRONIZATION

In scientific applications, often the communication occurs among a subset of processes like near-neighbourcommunication, ghost cell updates etc. For such scenarios a collective synchronization is semantically more easier to use as well more efficient to implement in the library. In this work, shown in the highlighted part of Figure 6.1 of the proposed research framework, we look at the various methods and algorithms to implement fence synchronization and provide an improved design and study the trade-offs.

Fence is an active synchronization method which is collective over the communicator associated with the window object. Fig. 6.2 shows a typical fence usage scenario. The first fence call makes sure that the window on the remote process is ready to be accessed. A process may issue one-sided operations after the first call to fence returns. The next call to fence or the second fence completes the one-sided operations issued by this process as well as the operations targeted at this process by other processes. An implementation of fence synchronization must support the following semantics: A one-sided operation cannot access a process's window until that process has called fence, and the second fence on a process cannot return until all processes needing to

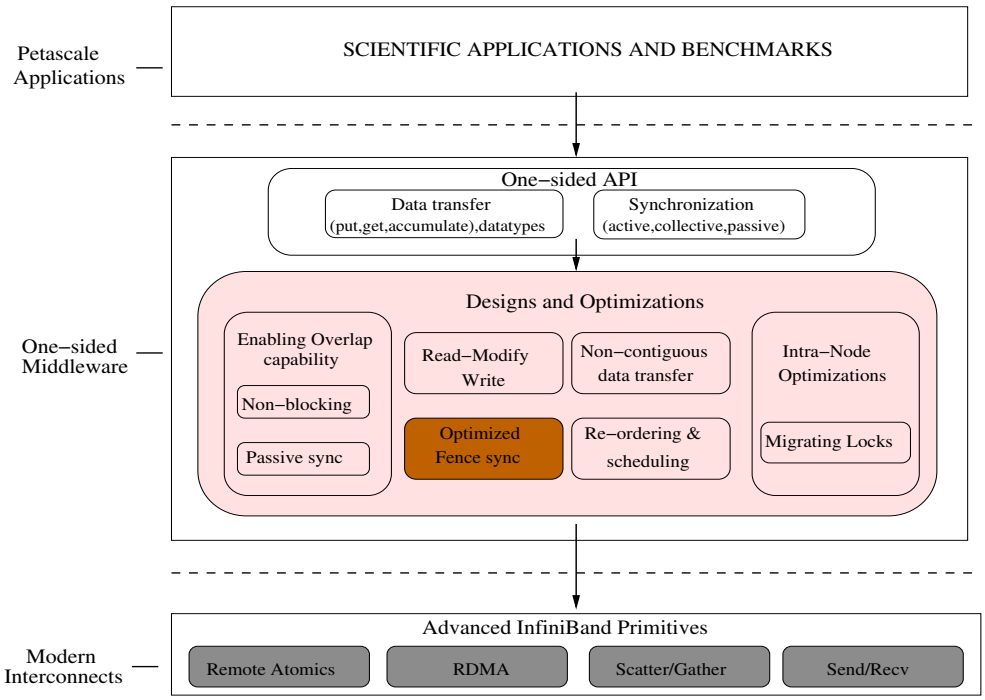

Figure 6.1: Overview

access that process's window have completed doing so. In addition the second fence also needs to start the next access epoch as seen in Fig. 6.2.

6.1 Design Alternatives

In this section we discuss the design choices for implementing fence mechanisms, identify the limitations and propose our optimizations.

In the MPI implementations derived from MPICH2 [9, 46, 60], there are two options for implementing fence: i) Deferred and ii) Immediate. In the Deferred approach, all the operations and synchronizations are deferred till the subsequent fence. In the Immediate method, the synchronization and communication operations happen as they are issued. We explore the design issues involved in both these approaches.

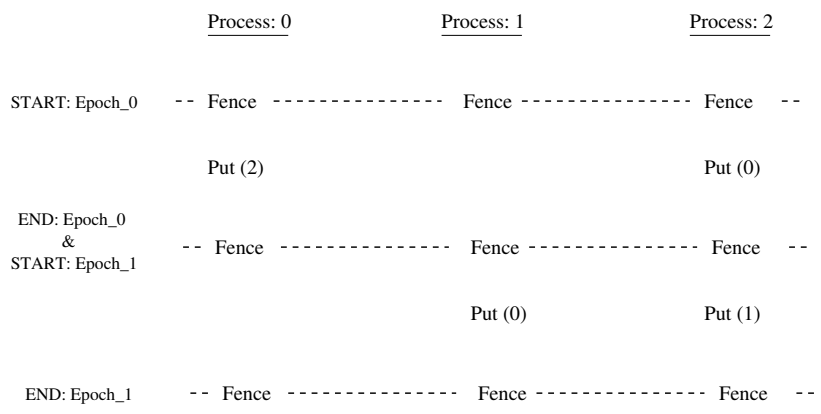

Figure 6.2: Fence Usage

As described in the previous section, a fence call needs to provide two functionalities: (i) it completes the previous epoch i.e it ensures that all the preceding RMA operations have completed and (ii) it begins the next exposure epoch.

Next we describe the design for implementing fence using the Deferred Approach.

6.2 Deferred Method using two-sided communication (Fence-Def)

In this design, the first fence call does nothing and returns immediately. All the ensuing one sided operations are queued up locally. All the work is done in the second fence, where each process goes through its list of queued operations to determine its target processes. This information is stored in an array and in the second fence operation a MPI_Reduce_scatter operation is performed to let every other process know if it is the target of RMA operations from this process. The remote process can then wait for the RMA operations from these nodes. The last RMA operation from each process is conveyed to the remote process by setting a flag in that RMA message.

Since the deferred approach is based on two-sided, the remote process is involved in receiving the RMA message and by looking at the flag, it ensures that it has received all the messages from that process. Since all the RMA messages are queued and issued during the fence, certain optimizations can be done that can improve the latency of the messages as well as reduce the overhead of the fence operations. However, there is no scope for providing overlap using this approach. In this design there is a notion of a remote agent that can handle incoming one-sided and synchronization messages and we refer to this two sided based design as *Fence-Def*.

6.3 Immediate Method using RDMA Semantics

Next we discuss fence implementations that uses immediate approach and RDMA semantics of the interconnects for communication operations. This is the main focus of our work since we are interested in fence implementation on networks that support RDMA semantics.

One of the main challenges in designing fence for RDMA operations is the detection of remote completion of the Put operations.

One approach to handle remote completion is to wait for local completions and then issue a Barrier operation. This seems perfectly plausible as the Barrier is called after all the Puts are issued and completed. However this does not completely guarantee correctness as shown in Fig. 6.3. There is scope for the Barrier messages to overtake the Put messages issued to process 3 as the Barrier can be implemented in a hierarchical fashion and can complete earlier than the Put. If there is a hardware implementation of Barrier and the underlying hardware guarantees that the messages are not overtaken, only then this is a valid solution but not otherwise.

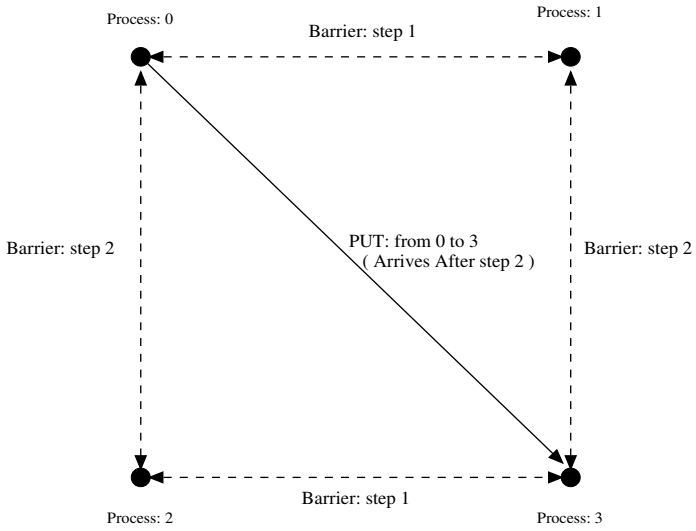

Figure 6.3: Barrier Messages overtaking Put

Another method of handling remote completion is by sending completion or finish notification messages that indicate that all messages on this channel have been received. There are some limitations of this approach with increasing scale.

6.3.1 Basic Design for Fence (Fence-Imm-Naive)

The MVAPICH2 library takes advantage of RDMA Read and Write operations to improve the performance of contiguous Get and Put operations. These one-sided operations are issued immediately. The one-sided based implementation provides higher bandwidth for large put and get messages than the two sided based design (Deferred method) and also provides greater potential for overlap of computation and communication. The current fence implementation is based on this design and is shown in Fig. 6.4. In order to completely implement the fence usage semantics shown earlier in Fig. 6.2, we need to support the following two functionalities: i) ensure

local and remote completion of operations in the current epoch and ii) indicate the beginning of the next access epoch.

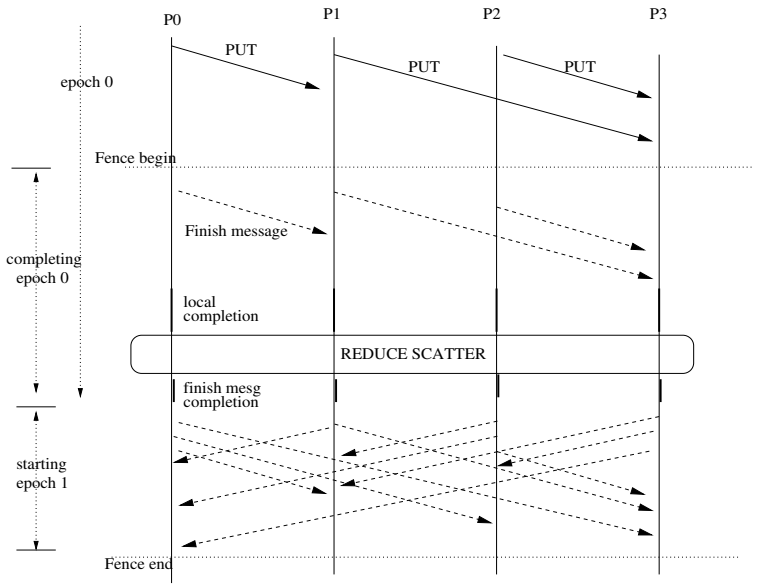

Figure 6.4: Fence-Imm-Naive

In this approach, polling for local completions are done to make sure that the issued one-sided operations are completed locally. For Get operations which are implemented on top of RDMA Read, local completion is sufficient to indicate that the Get operations are complete. The Put operations which are based on RDMA Write need remote completions. To handle this, a finish message is sent on each channel on which a put operation is issued to indicate that it has sent all the messages on that channel. Since the RDMA write operations on the same channel are ordered, when the finish message is received, all the RMA operations issued previously to that node are assured to be completed. Polling for local completions is done to make sure

that all the messages sent have completed locally. A Reduce_scatter operation is used to let a process know if it is the target of RMA operations. The target node then waits for finish messages from all these nodes. At this point, the fence has finished completion of messages for that epoch. The next part is to indicate to all the other processes that the next epoch can begin and it is safe to access the window. The current design posts a flag to every other process to indicate that the window can now be safely accessed for the next epoch.This results in all pair-wise synchronization of the processes. This is a naive approach and leads to flood of messages in the network. We will refer to this approach as *Fence-Imm-Naive*.

This design has several drawbacks that need to be addressed. From the description of the design in the previous section, we can see that there could be two potential floods of messages during the fence. The first is a flood of finish messages to handle remote completion if the process is communicating with several peers. The second flood is the flood of messages to post a flag to indicate that the window can be accessed for the next epoch.

6.3.2 Fence Immediate with Optimization (Fence-Imm-Opt)

As an optimization to this approach, we use a barrier instead of the pair-wise synchronization to indicate the beginning of the next epoch. This alleviates the second flood of messages described above. Figure 6.5 describes this approach and is a more scalable solution since it uses O(logn) communication steps. We refer to this approach as *Fence-Imm-Opt*.

These approaches described above still have the issue of completion messages being sent on all the channels. As the number of processes scale to large number,

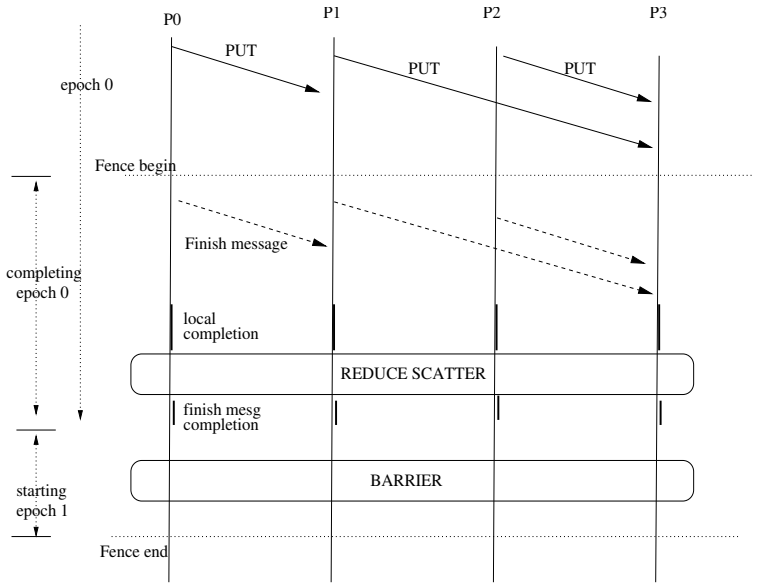

Figure 6.5: Optimized Design (Fence-Imm-Opt)

this could become a bottleneck. We propose a new design that uses the remote notification provided by the InfiniBand networks to design a novel and scalable fence implementation.

6.3.3 New Scalable Fence Design With Remote Notification (Fence-Imm-RI)

In this section we describe our new scheme which is also an Immediate method, but offers greater scalability. The new fence implementation is shown in Fig. 6.6. The main design and implementations issues are as follows:

Remote notification of one-sided operations

As described earlier, one approach to handle remote notifications is by flushing all the channels using a finish message. However, this approach is not scalable as

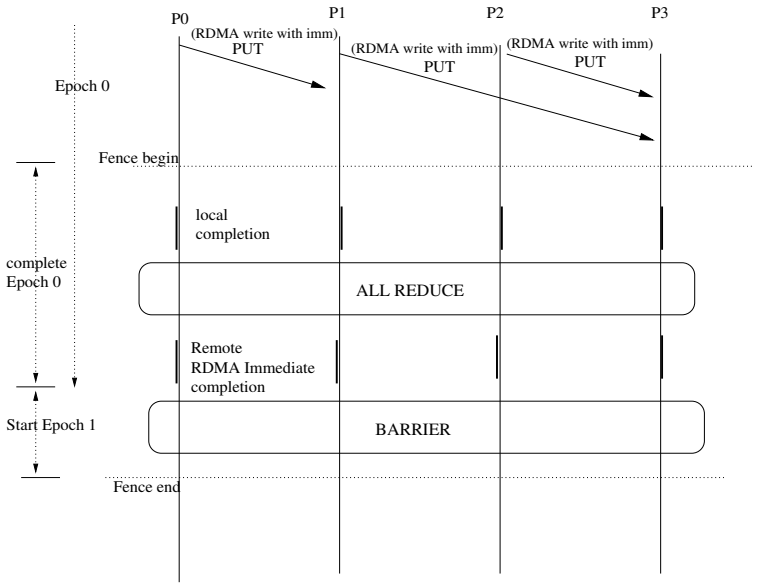

Figure 6.6: New design (Fence-Imm-RI)

it could lead to a flood of messages. In this design, we use the RDMA Write with Immediate operations to issue Put operations which creates a completion entry on the remote node. After polling for local completions, the remote node is informed of the number of such operations from all the processes through an MPI_AllReduce call. The remote node then polls till it receives completion notifications for that many number of RDMA write with Immediate operations. The completion of the Get operations is handled by waiting for local completions for the RDMA Read operations. This eliminates the first flood of messages.

Notification of beginning of next epoch

The next part is to indicate the beginning of the next epoch, i.e, to make sure that it is safe to access the window for the next epoch. It is to be noted that MPI

calls provide assertions that can be used to give hints if there are no preceding or succeeding one-sided operations and in that case the fence can be optimized. Here we do not handle the assertions, but look at the general case. In our design, we use a MPI_Barrier call to indicate the beginning of the next epoch. As mentioned earlier, typical Barrier implementation uses log(n) communication steps leading to a scalable solution. One trade-off of using this approach is that it forces everyone in the group to synchronize and we might lose out on some finer grain synchronization between a subset of members of the fence group.

Preposting Receive Descriptors

One issue with using RDMA Write with immediate functionality is the need to prepost receiver descriptors. We currently handle this issue by preposting a fixed number of receive descriptors initially and repost additional descriptors in the fence synchronization call. We post additional receives on receiving RDMA write completions. However, in cases where the fence synchronization is not called often and there are extremely large number of Put operations, there is a scenario in which we might run out of receive descriptors. One solution to this approach is to use the InfiniBand Shared Receive Queue (SRQ) mechanism [57] which allows efficient sharing of receive buffers across many connections. When the number of available buffers in the shared queue drops below a low watermark threshold, an interrupt can be generated and additional buffers are posted. Another approach is to use an asynchronous thread that can post the receives.

Henceforth we will refer to this approach as *Fence-Imm-RI*. In this work we have focused on InfiniBand Architecture. However, similar designs can be proposed for

other interconnects that can provide remote completion mechanisms for RDMA operations.

6.4 Experimental Results

In this section we present the experimental evaluation of the different fence designs. We characterize the performance of the proposed designs with the different micro-benchmarks representing various communication patterns.

Experimental testbed

Our experimental testbed is a 64 node (512-core) Intel cluster. Each node of our testbed is a dual processor (2.33 GHz quad-core) system with 4 GB main memory. The CPUs support the EM64T technology and run in 64 bit mode. The nodes support 8x PCI Express interfaces and are equipped with MT25208 HCAs with PCI Express interfaces. A Silverstorm 144 port switch is used to connect all the nodes. The operating system used is RedHat Linux AS4. All the experiments are run with one process per node configuration.

Methodology

In this section we describe the methodology for our evaluation. First we demonstrate the overlap capabilities of one sided based implementations as compared to one sided communication over two sided based implementations. Next, we focus on the synchronization overhead of our new Fence-Imm-RI design comparing it with implementations through a set of micro benchmarks and finally we compare the different designs for a Halo communcation pattern benchmark.

6.4.1 Overlap

In this section we demonstrate the overlap potential for our one-sided immediate approaches compared with the two sided implementation. Each process issues Put calls to its neighbor between two fence synchronization calls. Increasing amount of computation is inserted after the Put call and before the second fence call. The overlap is measured as the amount of computation that can be inserted without affecting the overall latency. The experiment was run for varying message sizes. The results are shown in Fig. 6.7. We observe that the two sided Deferred implementation shows virtually no overlap. This is expected because all the Put operations are deferred and issued inside the second fence and hence there is no scope for overlap. Whereas for all the Immediate approaches using one sided implementation good overlap can be achieved for message sizes beyond 16K and close to 90% overlap for message sizes larger than 64k. In the following sections we concentrate on comparing the synchronization overhead of our new fence design (Fence-Imm-RI) as compared to all the other approaches.

6.4.2 Basic Collectives Performance

Since the fence designs use some of the collectives in its implementation in order to exchange the number of remote operations as well as to synchronize for the next epoch, we show the baseline performance of the collective operations: Barrier,All-Reduce and Reduce_scatter first in this section. This would help us in understanding the performance of various fence designs. Table 6.1 shows the results for up to 64 processes, for these collectives. The All-Reduce and Reduce_scatter numbers are

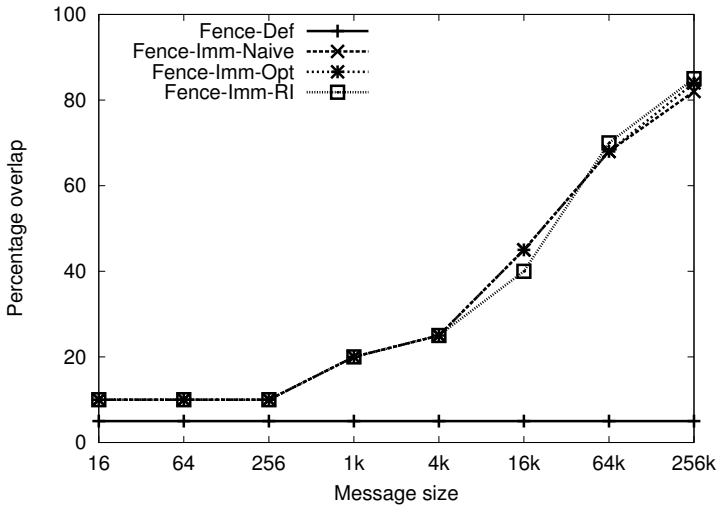

Figure 6.7: Overlap performance

shown for 256 bytes message size. These collectives show good scalability with 40-50 usecs latencies on 64 processes.

Numprocs	Barrier	Allreduce	Reduce_Scatter
2	3.66	7.75	6.84
4	10.79	13.78	11.27
8	18.65	20.9	16.26
16	27.21	30.34	21.99
32	37.89	43.15	29.19
64	44.13	51.9	33.18

Table 6.1: Basic Collectives Performance (usecs)

6.4.3 Fence Synchronization Performance

In this section we evaluate the performance of the fence alone without any one-sided communication operations. This measures the overhead involved in a fence

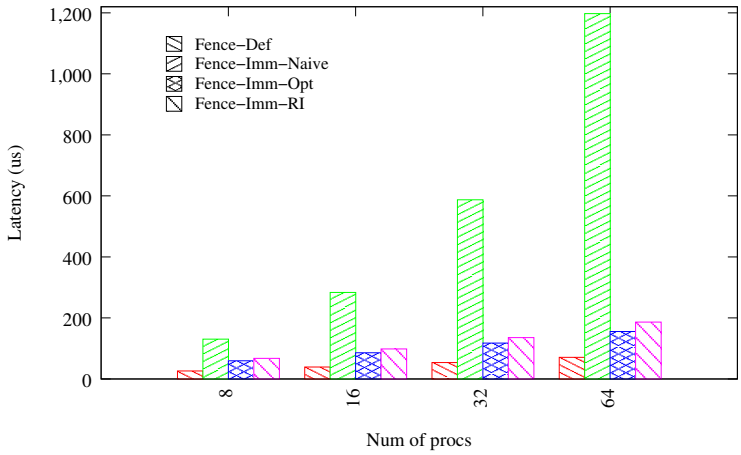

Figure 6.8: Fence Performance for Zero Put

synchronization. The results are shown in Fig. 6.8. Since there are no data transfer operations, there is no overhead of the data messages in terms of local and remote completions for one-sided operations. We still need to use the collectives to inform the other processes that the fence can complete and also that the next fence epoch can begin. The Fence-Imm-Naive performs the worst, because of the all pair-wise synchronization happening to indicate the end of the epoch. The Fence-Imm-Opt and Fence-Imm-RI perform close to each other since both of them use Barrier to indicate the start of next epoch. The Fence-Imm-Opt performs slightly better than the Fence-Imm-RI, the reason for this is because the Fence-Imm-Opt uses Reduce Scatter collective as opposed to the AllReduce collective used by the Fence-Imm-RI scheme. From Table 6.1, we can see that the Reduce Scatter collective has a lower latency than that of AllReduce. We see that the Fence-Def which uses the two sided approach performs the best, since it does not need to use additional collective to indicate the start of an epoch.

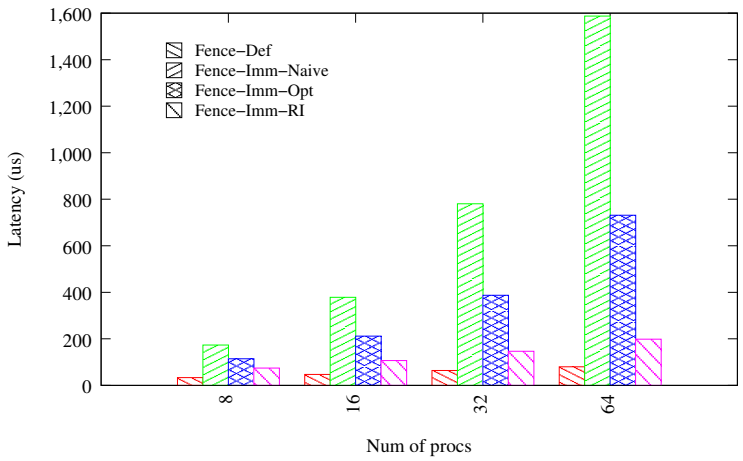

Figure 6.9: Fence Performance for Single Put

6.4.4 Fence Synchronization with Communication Performance

In the previous section, we evaluated the different schemes for just the fence synchronization overhead. In this section we evaluate the scalability of our fence implementations with communication operations which is more reflective of usage in a one-sided application. First we evaluate the performance of fence with a single Put of 16 bytes message size issued by all the processes. The results are shown in Fig. 6.9. For this pattern, we observe that Fence-Imm-Naive performs very badly. However it is interesting to compare the performance of Fence-Imm-Opt and Fence-Imm-RI. We now see that the Fence-Imm-RI outperforms the Fence-Imm-Opt scheme. The reason for this is the Fence-Imm-RI relies on the hardware RDMA-Write with immediate for remote completions, whereas the Fence-Imm-Opt has to issue completion messages which increases the overhead. This difference is magnified further in the next experiment where each process issues Puts to 8 neighbors and hence the number of completion messages increases further for the Fence-Imm-Opt. The results for this

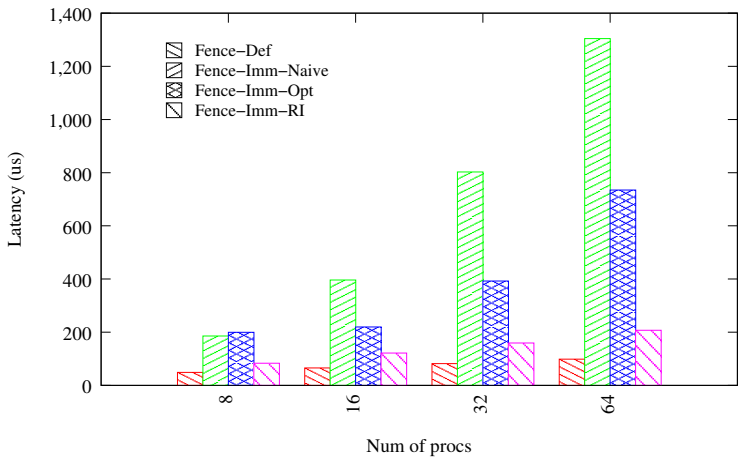

Figure 6.10: Fence Performance for Multiple Puts

experiment is shown in Fig. 6.10. The two-sided approach still performs the best because it has lower overhead for small messages and can combine the data transfer and synchronization message. But it needs to be noted that it has poor overlap capability.

6.4.5 Halo Exchange Communication Pattern

Scientific applications often communicate in a regular pattern. Halo exchange of messages is a very popular model in which each node communicates with a fixed number (4, 8, 26, etc) of neighbors. These usually correspond to the parallel processing of multi-dimensional data in which each compute process handles a certain section of this data set. The neighbors exchange messages to handle border conditions. This communication pattern is more representative of real world applications. We simulate this halo exchange pattern for 4 and 8 neighbors and evaluate the two schemes. Every process initiates the one-sided operation with its neighbor and simultaneously performs a fixed amount of computation.

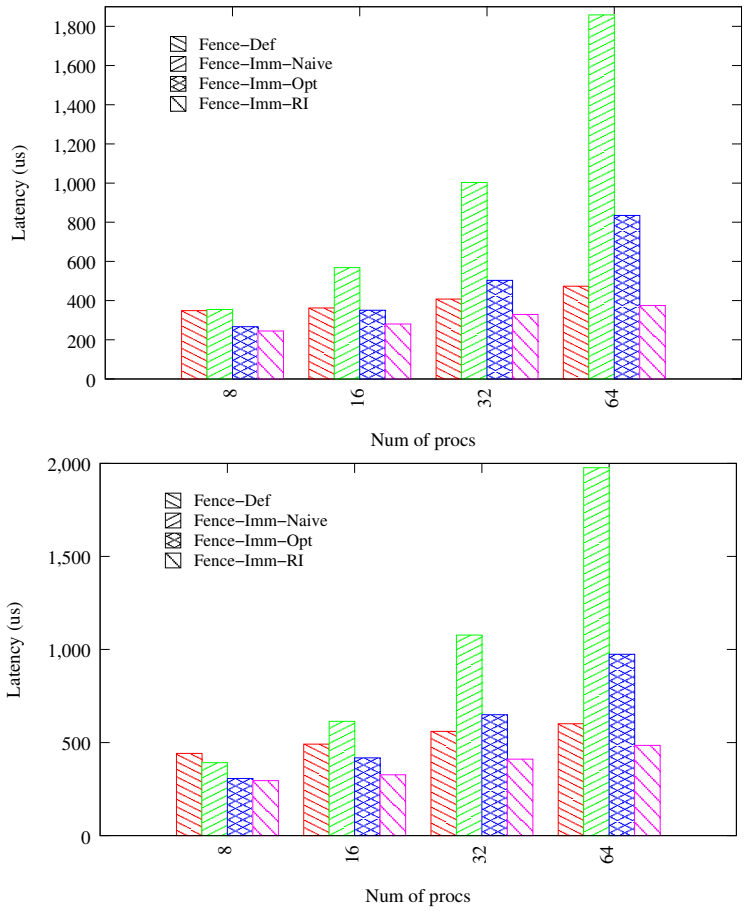

Figure 6.11: Fence performance with Halo Exchange: (a) 4 neighbors and (b) 8 neighbors

The results for 4 and 8 neighbors are shown in Fig. 6.11(a) and Fig. 6.11(b), respectively. Here we observe that our new Fence-Imm-RI scheme outperforms all the other schemes. All the immediate approaches have good computation/ communication overlap, whereas the two-sided deferred approach has very poor computation/communication overlap. The Fence-Imm-RI has reasonably low synchronization overhead and very good computation/communication overlap and hence shows the best performance.

6.5 Related Work

Some of the MPI-2 implementations that support one-sided communication are MPICH2 [9, 33], OpenMPI [12],WMPI [44], NEC [63], SUN-MPI [16]. The NEC implementation [63] uses Allreduce and Barrier to implement fence synchronization. However they do not use RDMA Write with Immediate mechanism for remote notifications. The RDMA Write with Immediate feature has been explored in [42] for designing MPI_Alltoall over InfiniBand. In our work we are using it to design a scalable fence synchronization.

CHAPTER 7

READ MODIFY WRITE MECHANISMS

One of the important operations in a one-sided model is *read-modify-write*. Applications like Hydra[51] which is based on MPI-2 one-sided, predominantly use this operation. One-sided applications can either use these interface if they are provided, else they need to build on top of existing primitives. MPI-2 semantics provide MPI_Put, MPI_Get and MPI_Accumulate operations that can be used to implement the *read-modify-write* operations. In this work, shown in the highlighted part of Figure 7.1 of the proposed research framework, we study the different mechansims for providing this capability and further explore how the remote atomic operations provided by InfiniBand can be leveraged to provide better support for these operations.

7.1 HPCC Benchmark

HPCC Benchmark suite is a set of tests that examine the performance of HPC architectures that stress different aspects of HPC systems involving memory and network in addition to computation [56]. HPCC Random Access benchmark is one of the benchmarks in this suite which measures the rate of random updates to remote memory locations. Currently this benchmark is implemented based on MPI two-sided semantics. In this work we design different MPI-2 versions of the Random

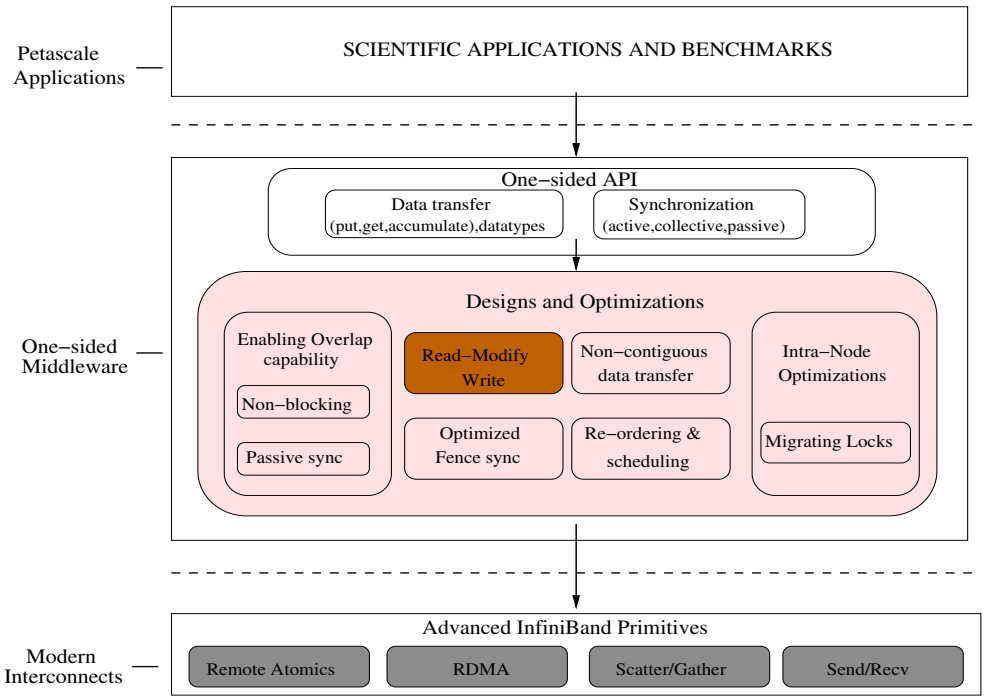

Figure 7.1: Overview

Access benchmark using the MPI-2 one-sided alternatives. We use the one-sided versions of the Random Access benchmark as a case study for studying different implementations of the *read-modify-write* operations and provide optimizations to improve the performance.

The HPC Challenge (HPCC) benchmark suite has been funded by the DARPA High Productivity Computing Systems (HPCS) program to help define the performance boundaries of future Petascale computing systems [22]. HPCC is a suite of tests that examine the performance of high-end architectures using kernels with memory access patterns more challenging than those of the High Performance LINPACK (HPL) benchmark used in the Top500 list. The Random Access benchmark measures

the rate of integer updates to random memory locations (GUPs). It uses xor operation to perform the updates on the remote node. The verification procedure allows 1% incorrect or skipped updates which allows loose concurrent memory update semantics on shared memory architecture. It allows optimization in terms of aggregating up to 1024 updates to improve the performance. There has been earlier work to improve the performance of this benchmark for blue-gene clusters [26].

7.2 One sided HPCC Random Access Benchmark: Design Alternatives

In this section we describe the different approaches taken to implement the one sided version of the HPCC Random Access benchmark. As described earlier, the random access benchmark measures the GUPs rating. The term randomly means that there is little relationship between one address to be updated and the next. An update is a read-modify-write operation on a table of 64-bit words. An address is generated, the value at that address read from memory, modified by an xor operation with a literal value and that new value is written back to memory. Currently the MPI version of the benchmark is based on two sided version. In this version the random address and value is generated and is sent to the remote node. The remote node receives this data and appropriately updates the memory location.

Design Issues

In this section we first describe the semantics and mechanisms offered by MPI-2 for designing one-sided applications. In a one-sided model, the sender can access the remote address space directly without an explicit receive posted by the remote node. The memory area on the target process that can be accessed by the origin process

is called a Window. In this model we have the communication operations MPI_Put, MPI_Get and MPI_Accumulate and the synchronization calls to make sure that the issued one sided operations are complete. There are two types of synchronization: a) active in which the remote node is involved and b) passive in which the remote node is not involved in the synchronization. The active synchronization calls are collective on the entire group in case of MPI_Fence or a smaller group in case of Start_Complete and Post_Wait model. This could lead to some limitations when the number of synchronizations needed per process are different for different nodes. In passive synchronization the origin process issues MPI_Lock and MPI_Unlock call to indicate the beginning and end of the access epoch. Next we describe our approach taken in designing the one-sided versions of the HPCC Random Access benchmark. We map the table memory to the Window so that the one-sided versions can read and write directly to this memory.

7.2.1 HPCC Get-Modify-Put (HPCC_GMP)

In the first approach we call MPI_Get to get the data, perform the modification, then use MPI_Put to put the updated data to the remote location. As compared to the two sided versions there are no receive calls made on the remote node. Also the active synchronization model cannot be used since we cannot match the number of synchronization calls across all nodes. This is because the number of remote updates as well as the location of the remote updates for each node can vary randomly. Hence we use passive synchronization MPI_Lock and MPI_Unlock calls in this scheme. Further we need one set of Lock and Unlock calls to fetch the data, perform the modification, then another set of Lock and Unlock operations to put the data. The

reason for this is the flexibility of MPI-2 semantics which allows MPI_Get to fetch the data in Unlock. Also the MPI_Get and MPI_Put can be reordered within an access epoch. We describe this approach in Fig. 7.2a and will henceforth refer to it as *HPCC_GMP*. This approach leads to a lot of network operations resulting in lower performance. Further the possibility of incorrect updates increases. This is due to the coherency issues that might arise because of parallel updates occurring simultaneously. To make sure that there are no incorrect updates, mutual exclusion (atomicity) has to be implemented on top of the existing approach which could lead to further degradation in performance.

7.2.2 HPCC Accumulate (HPCC_ACC)

Our next approach uses the MPI_Accumulate operation provided by MPI-2. MPI-2 semantics provide MPI_Accumulate which are basically atomic reductions. This non collective one-sided operation combines communication and computation in a single interface. It allows the programmer to update atomically remote locations by combining the content of the local buffer with the remote memory buffer. This implementation calls MPI_Accumulate between MPI_Lock and MPI_Unlock synchronization calls. Using this approach shown in Fig. 7.2b, we do not have the issue of incorrect updates. Also as compared to our *HPCC_GMP*, the number of network operations is significantly reduced. Another approach is to use Accumulate with Active synchronization model using Win_Fence. This could be done by calling Win_Fence at the very beginning, performing all the updates using MPI_Accumulate and then call one Win_Fence at the very end. All the processes need to call two Win_Fence

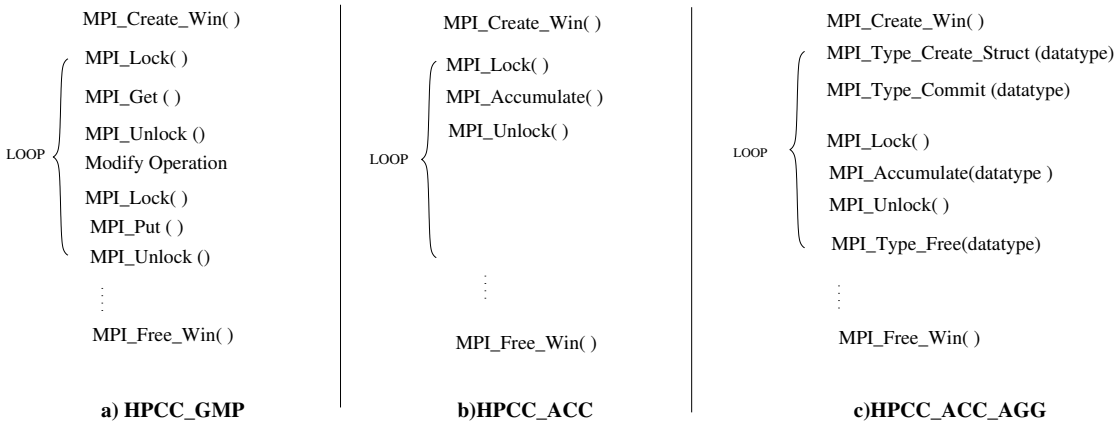

Figure 7.2: Code snippets of one-sided versions of HPCC Random Access benchmark

calls, one at the beginning and one at the end. However since MPI-2 semantics allows the actual data transfer to occur inside the synchronization call that closes the exposure epoch, all the accumulates could happen during the second Win_Fence call. Many MPI implementations actually make use of this flexibility. This violates the random benchmark rule that you could store only 1024 updates at the maximum before sending them. Hence we did not consider this approach.

7.3 Optimizations

In this section we describe two optimizations we propose in this paper to improve the performance of the one-sided version of HPCC Random Access benchmark.

7.3.1 Software Aggregation

In this technique we want to aggregate or pack a number of update operations together so that the overhead of sending as well as synchronization operations can

be reduced. Using this approach, we aggregate a bunch of update operations before sending them as a single communication operation. The HPCC random access benchmark allows each processor to store up to 1024 updates before sending them out. The MPI-2 semantics provides *datatypes* feature that can be leveraged to achieve aggregation. For one-sided operations both the sender and destination datatypes need to be created. We create MPI_Type_struct sender and receiver datatypes to represent a bunch of updates in the following manner. The count holds the number of updates to be aggregated, the block_lengths are all one, the displacement array holds the remote address or local address respectively of each update and the MPI datatype of each entry is 64 bit unsigned integer. We then use the created datatypes to issue a single communication call as shown in Fig. 7.2c. Using this approach we expect to improve the performance since the number of network operations are minimized.

7.3.2 Hardware based Direct Accumulate

InfiniBand provides hardware atomic fetch and add operation that can be leveraged to optimize MPI_Accumulate operation for MPI_SUM. The Accumulate operations use the hardware fetch and add operation that can provide good latency and scalability. One of the limitations of this approach is that we can only do single 64 bit accumulates with each fetch and add operation, i.e. aggregation is not possible. A benefit of using this approach is that since it is truly one-sided in nature, it provides more scope for overlap that can lead to improved performance. It is to be noted that this optimization is implemented in the underlying MVAPICH2 MPI library as a prototype and is transparent to the application writer.

7.4 Performance Evaluation

In this section, we evaluate the performance of the one-sided version of the HPCC benchmark for the different schemes. We present some micro-benchmark results to give the basic performance of different one-sided operations and show the potential of our proposed optimizations. The experimental testbed is x86 64 node cluster with 32 Opteron nodes and 32 Intel nodes. Each node has 4GB memory and equipped with PCI-Express interface and InfiniBand DDR network adapters (Mellanox InfiniHost III Ex HCA).

Basic performance of one-sided operations

In this section we show the performance of the basic one-sided operations MPI_Put, MPI_Get and MPI_Accumulate. Fig. 7.3a shows the small message latency for these operations. The latency for 8bytes for put and get are 5.68 and 11.03 usecs, respectively, whereas the accumulate latency is 7.06 usecs. Since get_modify_put implementation needs both get and put in addition to modify and synchronization operation, we expect this performance to be lower compared to the accumulate based approach.

HPCC one-sided benchmark performance with different schemes

In this section we evaluate the performance of the two different versions of the benchmark *HPCC_GMP* and *HPCC_ACC*. The results are shown in Fig. 7.3b. As expected the *HPCC_ACC* performs better than the *HPCC_GMP* because of the number of synchronization and communication operations in *HPCC_GMP*. The overhead of these additional network operations leads to lower performance of *HPCC_GMP*.

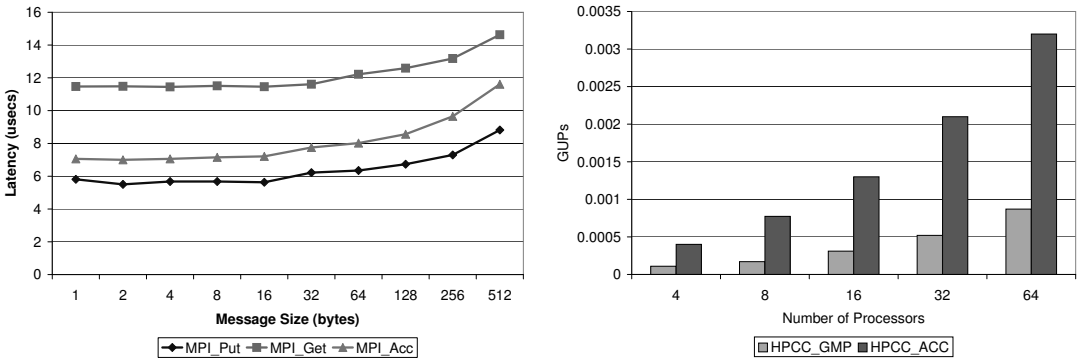

Figure 7.3: Basic Performance (a) Micro-benchmarks and (b) Basic HPCC GUPs

This performance gap increases with increasing number of processors since the synchronization cost increases further for larger number of nodes. Hence we choose *HPCC_ACC* as our base case for further optimizations and evaluations.

Aggregation Benefits

To improve the performance of the Accumulate operation, we proposed aggregation using Accumulate with datatype. In this section we evaluate the performance benefits of using datatype at micro-benchmark level. In the basic version we do multiple accumulates corresponding to the number of updates. In the aggregated version we create a datatype corresponding to the number of updates and perform a single accumulate operation with that datatype. Fig. 7.4a shows the results of our study. With increasing amounts of aggregation, the Accumulate with datatype outperforms the multiple accumulate schemes. With aggregation the cost of sending overhead and the synchronization overheads are limited to the number of aggregated operations. Next we compare the performance of *HPCC_ACC_AGG* with *HPCC_ACC* for 512 and 1024 aggregations. The results are shown in Fig. 7.4b. We observe a similar

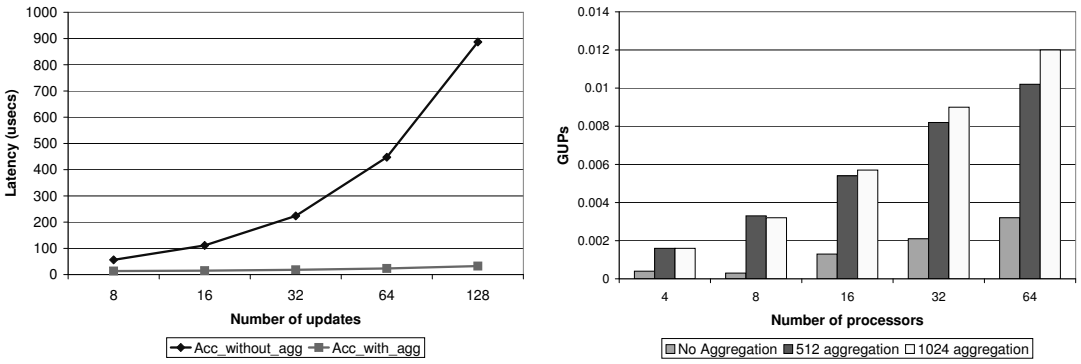

Figure 7.4: Aggregation Performance Benefits (a) Basic Aggregation Microbenchmarks and (b) HPCC with Aggregation

trend with the optimized *HPCC_ACC_AGG* performing better than the *HPCC_ACC* scheme. This result demonstrates the benefits that aggregation can provide.

Hardware based Direct Accumulate

In this section we first study the benefits that could be achieved using the hardware based fetch and add operation to implement a read modify write operation at microbenchmark level (*DIRECT_ACC*). We compare its performance with the the schemes that uses Get Modify Put (*GMP*) approach and MPI_Accumulate (*ACC*) approach. The MPI implementation allows optimizations that delays the actual lock and data transfer operation to happen during unlock. In this case measuring just the lock and unlock cost does not provide any additional insight. Hence we measure the latency that includes both data transfer and lock/unlock synchronization operation. Fig. 7.5 compares the basic performance of *GMP*, *ACC* and *DIRECT_ACC*. We note that for single updates of 64bit integer, the (*DIRECT_ACC*) scheme provides the

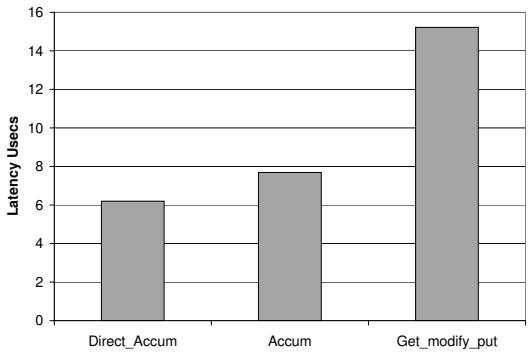

Figure 7.5: Direct Accumulate Performance Benefits: Micro-benchmarks

lowest latency. This is because the existing MPI_Accumulate implementation is inherently two sided whereas the Direct Accumulate implementation makes use of the truly one-sided hardware feature.

Next we try to understand the benefits that a hardware based Accumulate operation can provide to an application. To evaluate this we modify the *HPCC_ACC* benchmark to use the MPI_SUM operation instead of the MPI_BXOR operation and call this as *HPCC_ACC_MOD*. The verification phase is correspondingly modified. We then compare the *HPCC_ACC* which uses the existing MPI_Accumulate implementation in the MVAPICH2 library with the modified *HPCC_ACC_MOD* which uses our Direct Accumulate prototype implementation. The results are shown in Fig. 7.6. We observe that the Direct accumulate performs significantly better than the basic accumulate. Also the Direct Accumulate seems to scale very well with increasing number of processors. The reason for this is two-fold: 1) low software overhead and 2) true one-sided nature of the hardware based Direct Accumulate.

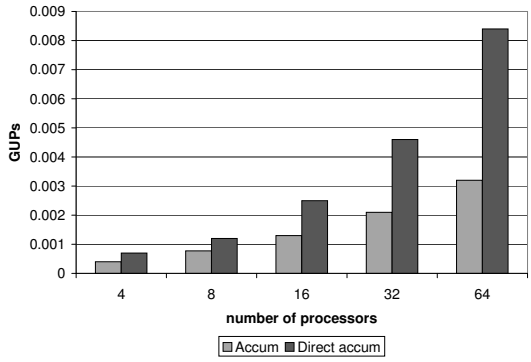

Figure 7.6: HPCC with Direct Accumulate

Finally we compare our two proposed techniques Direct Accumulate and software aggregation (Accumulate with datatype). The results are shown in Fig. 7.7. The software aggregation scheme beats the hardware based direct accumulate approach since currently the hardware fetch and add operation does not support aggregation. Also the gap between the two schemes seem to be narrowing with increasing nodes. This demonstrates the scalability of the hardware based operations and suggests the benefits of having aggregation in hardware as well.

In this work, we designed MPI-2 one-sided versions of HPCC random access benchmark using get_modify_put and MPI_Accumulate operations. The modified one-sided HPCC Random Access benchmarks are available on line for reference [13]. We evaluated these two different approaches on a 64 node cluster. To improve the performance we explored two different techniques: a) software based aggregation and b) utilizing hardware atomic operations. We analyzed the benefits and trade-offs of these two

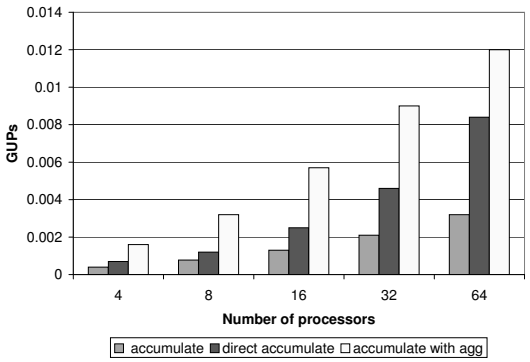

Figure 7.7: Software Aggregation vs Hardware Direct Accumulate benefits

approaches. Our studies show that the software based aggregation performs the best. We also demonstrated the potential and scalability of the hardware based approach.

7.4.1 Discussion

Current implementations for HPCC Random Access benchmark are based on two-sided communication primitives. While the main objective of this work is not to compare the designs based on one-sided and two-sided semantics, it is also important in this context to note that the current one-sided implementations are largely based on two-sided primitives in the MPI libraries and hence, such an evaluation is not as informative. InfiniBand's hardware fetch and add operation provides a design opportunity for a Direct Accumulate for MPI_Sum operation for a single 64 bit field. While we have demonstrated that both aggregation and direct hardware based accumulation has benefits, an aggregated direct accumulate is likely to yield much higher performance benefit. However it is clearly not possible to implement such a design with current InfiniBand's hardware. Also, it is to be noted that the hardware

fetch and add operation currently only allows the implementation of accumulation of MPI_Sum for 64 bit fields and other operations need additional hardware support.

7.5 Related Work

In [40, 37], the authors have used InfiniBand hardware features to optimize the performance of MPI-2 one sided operations. Other researchers [39] study the different approaches for implementing the one sided atomic reduction. The authors in [17] have looked at utilizing the hardware atomic operations in Myrinet/GM to implement efficient synchronization operations. Recently several researchers have been looking at providing optimizations to the HPCC benchmark. In [26] the authors have suggested techniques for optimizing the Random access benchmark for Blue Gene clusters. In [59] the authors have evaluated UPC programming model on Cray machines using the HPCC benchmark suite.

CHAPTER 8

NON-CONTIGUOUS DATA-TRANSFERS

Non-contiguous communication patterns are quite common in scientific applications. several MPI applications such as (de)composition of multi-dimensional data volumes [10, 24] and finite-element codes [18] often need to exchange data with algorithm-related layouts between two processes. In the NAS benchmarks such as MG, LU, BT, and SP, non-contiguous data communication has been found to be dominant [41]. As one of its important features, MPI provides datatype as a powerful and general way of describing arbitrary collections of data in memory in a compact fashion. The MPI standard also provides run time support to create and manage such MPI derived datatypes. MPI derived datatypes are expected to become a key aid in application development. In practice, however, the poor performance of many MPI implementations with derived datatypes [18, 32] becomes a barrier to using derived datatypes. This is primarily due to copy overhead associated with multiple copies from and to contiguous buffers internally.

A programmer often prefers packing and unpacking non contiguous data manually even with considerable effort. Recently, a significant amount of research work have concentrated on improving datatype communication in MPI implementations, including 1) Improved datatype processing system [32, 52], 2) Optimized packing

and unpacking procedures [18, 32], and 3) Taking advantage of network features to improve non contiguous data communication [67]. Our previous work used multiple RDMA writes, henceforth referred to as Multi-W, as an effective solution to achieve zero-copy datatype communication [67].

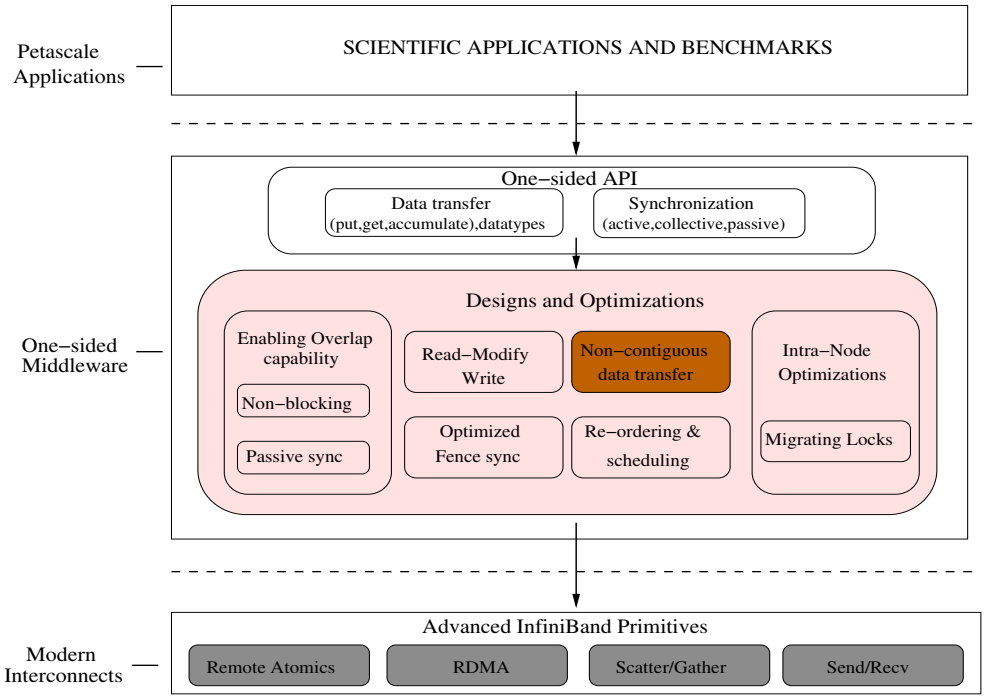

Figure 8.1: Overview

In this work, shown in the highlighted part of Figure 8.1 of the proposed research framework, we focus on improving non-contiguous data communication by taking advantage of advanced features of modern interconnects. The drawback of the traditional pack/unpack based approaches for implementing datatypes is that it involves memory copies on both sender and receiver sides. Thus, zero copy communication protocols are of increased importance because they improve memory performance

and also have reduced host CPU involvement in moving data. Hence we focus on leveraging the benefits of zero copy message transfers to implement efficient protocols for datatype communication. In this work we explore zero-copy designs using InfiniBand's hardware scatter/gather operations.

8.1 Non-contiguous Point-to-point Data-transfer

The motivation for proposing our new zero-copy scheme is two-fold. First, we would like to address/alleviate the limitations of our previous approaches. Secondly, with the emergence of PCI-Express bus, the network bandwidth that can be utilized is greatly enhanced. This further reinforces the need to come up with schemes that can directly exploit this enhanced bandwidth to the maximum. Zero copy schemes, because they are not limited by memory bandwidth are more appealing. However based on our previous work, though the Multi-W zero copy scheme does better than the copy based approaches, it still may result in under utilization of the network in many scenarios. InfiniBand provides the Gather Send and Scatter Receive capability through send/receive channel semantics. We would like to explore this option to come up with an efficient zero copy scheme. The following experiment below tries to assess the potential benefits of using Send Gather and Receive Scatter at the VAPI layer (low level InfiniBand API provided by Mellanox).

Motivating Case Study for the Proposed SGRS Scheme

Consider a case study involving the transfer of multiple columns in a two dimensional $M \times N$ integer array from one process to another. There are two possible zero-copy schemes. The first approach is to use multiple RDMA writes, one per row. The second approach uses Send Gather/Receive Scatter. We compare these two

schemes over the VAPI layer, which is an InfiniBand API provided by Mellanox [7]. The first scheme posts a list of RDMA write descriptors. Each descriptor writes one contiguous block in each row. The second scheme posts multiple Send Gather descriptors and Receiver Scatter descriptors. Each descriptor has 50 blocks from 50 different rows (50 is the maximum number of segments supported in one descriptor in the current version of Mellnox SDK). We will henceforth refer to these two schemes as "Multi-W" and "SGRS" in the plots. In the first test, we consider a 64×4096 integer array. The number of columns being sent varies from 8 to 2048. The total message size varies from 2 KBytes to 512 KBytes accordingly. The bandwidth test is used for evaluation and the bandwidth number is reported in order of Million bytes (MB/s). As shown in Figure 8.2, the SGRS scheme consistently outperforms the Multi-W scheme. In the second test, the number of blocks varies from 4 to 64. Three different

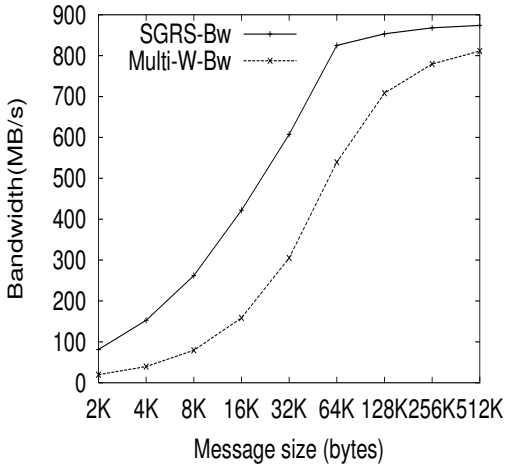

Figure 8.2: Bandwidth Comparison over VAPI with 64 Blocks

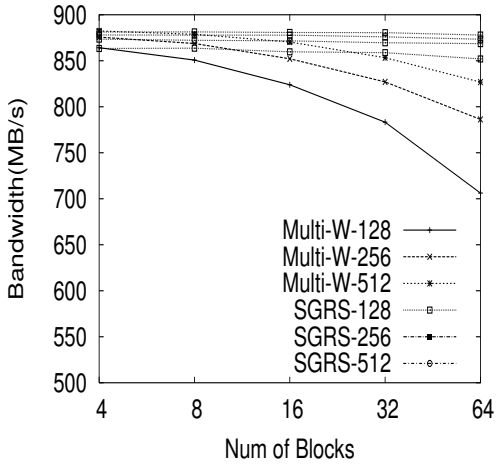

Figure 8.3: Bandwidth Comparison over VAPI with Varying Number of Blocks

message sizes were studied: 128 KBytes, 256 KBytes, and 512 KBytes. Figure 8.3

shows the bandwidth results with different number of blocks and different message sizes. When the number of blocks is small, both Multi-W and SGRS schemes perform comparably. This is because the block size is relatively large. The network utilization in the Multi-W is still high. As the number of segments increase we observe a significant fall in bandwidth for the Multi-W scheme whereas the fall in bandwidth is negligible for the SGRS scheme. There are two reasons. First, the network utilization becomes lower when the block size decreases (i.e. the number of blocks increases) in the Multi-W scheme. However, in the SGRS scheme, the multiple blocks in one send or receive descriptor are considered as one message. Second, the total startup costs in the Multi-W scheme increases with the increase of the number of blocks because each block is treated as an individual message in the Multi-W scheme and hence the startup cost is associated with each block. From these two examples, it can be observed that the SGRS scheme can overcome the two drawbacks in the Multi-W by increasing network utilization and reducing startup costs. These potential benefits motivate us to design MPI datatype communication using the SGRS scheme described in detail in Section 10.

8.1.1 Proposed SGRS (Send Gather/Recv Scatter) Approach

In this section we first describe the SGRS scheme. Then we discuss the design and implementation issues and finally look at some optimizations to this scheme. The basic idea behind the SGRS scheme is to use the scatter/gather feature associated with the send receive mechanism to achieve zero-copy communication. With this feature we can send/receive multiple data blocks as a single message by posting a send gather descriptor at source and a receive scatter descriptor at destination.

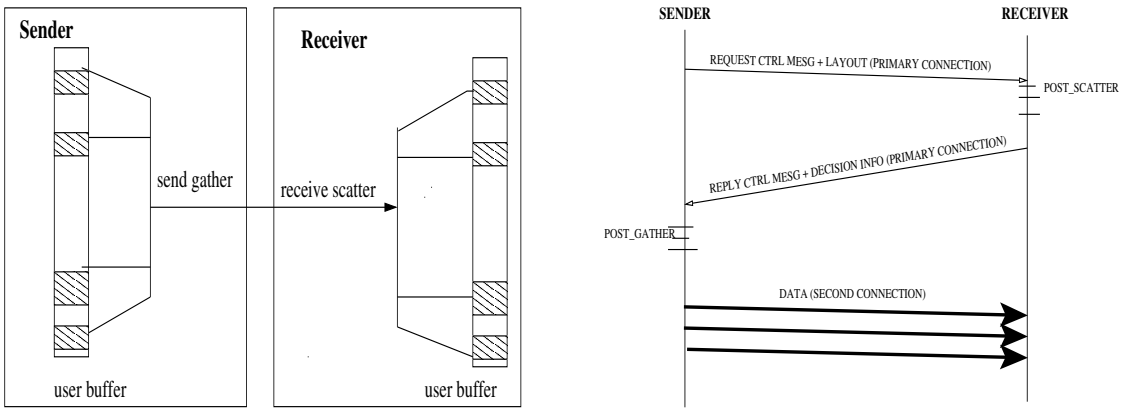

Figure 8.4: a)Basic Idea of the SGRS Scheme and b) SGRS Protocol.

Figure 8.4a illustrates this approach. InfiniBand also provides RDMA Write with Gather and RDMA Read with Scatter capability. The SGRS scheme can handle non-contiguity on both sides. The RDMA Write Gather or RDMA Read Scatter handles non-contiguity only on one side. Hence, to achieve zero-copy datatype communication based on RDMA operations, the Multi-W scheme is needed [67]. Compared to the Multi-W scheme, the SGRS scheme reduces the number of descriptors dramatically. It also increases the network utilization. There are two requirements. First, all the contiguous blocks need to be registered. Second, the sender should send its layout information to the receiver. The cost of sending the layout could be high in some cases. We describe optimization mechanisms like layout caching later in this section to alleviate this problem.

Design and Implementation Issues

We now discuss the intrinsic issues related to the MPI implementation of the SGRS scheme. The communication protocol and design issues such as secondary connection, progress, layout exchange, posting descriptors, and user buffer registration are addressed here.

Communication Protocol

The SGRS scheme is deployed in Rendezvous protocol to transfer large datatype messages. For small datatype messages, the Generic scheme is used. As shown in Figure 8.4b, the sender first sends the Rendezvous start message with the data layout information out. Second, the receiver receives the above message and figures out how to match the sender's layout with its own layout. Then, the receiver sends the layout matching decision to the sender. After receiving the reply message, the sender posts send gather descriptors. It is possible that the sender may break one block into multiple blocks to meet the layout matching decision. There are several design issues: Secondary connection, Progress, Layout exchange, Posting descriptors and Registration.

Secondary connection

The SGRS scheme needs a second connection to transmit the non-contiguous data. This need arises because it is possible in the existing MVAPICH design to prepost some receive descriptors on the main connection as a part of its flow control mechanism. These descriptors could unwittingly match with the gather-scatter descriptors associated with the non-contiguous transfer. One possible issue with the extra connection is scalability. In our design, there are no buffers/resources for the second

connection. The HCA usually can support a large number of connections. Hence the extra connection does not hurt the scalability. The second issue is out of order messages. Having two connections can create out of order arrival of messages which have to be handled carefully. However, in our design, since the control messages as shown in Figure 8.4b still use the primary connection, the out of order situation is averted and the receiver still receives the message in the same order.

Progress and Completion

Another issue is handling of completion of a message. In our design we associate a single completion queue with both connections. This fits in well with the existing framework for ensuring progress of the communication call. The completion is handled by polling for completion of scatter/gather descriptors on the second connection, and we do not need an extra message to indicate completion.

Layout exchange

The MPI datatype has only local semantics. To enable zero-copy communication, both sides should have an agreement on how to send and receive data. In our design, the sender first sends its layout information to the receiver in the Rendezvous start message as shown in Figure 8.4b. Then the receiver finds a solution to match these layouts. This decision information is also sent back to the sender for posting send gather descriptors. To reduce the overhead for transferring datatype layout information, a layout caching mechanism is desirable [36]. Implementation details of this cache mechanism in MVAPICH can be found in [67]. In Section 8.2, we evaluate the effectiveness of this cache mechanism.

Posting Descriptors

There are three issues in posting descriptors. First, if the number of blocks in the datatype message is larger than the maximum allowable gather/scatter limit, the message has to be chopped into multiple gather/scatter descriptors. Second, the number of posted send descriptors and the number of posted receive descriptors must be equal. Third, for each pair of matched send and receive descriptors, the data length must be the same. This basically needs a negotiation phase. Both these issues can be handled by taking advantage of the Rendezvous start and reply message in the Rendezvous protocol. In our design, the receiver makes the matching decision taking into account the layouts as well as scatter-gather limit. Both the sender and the receiver post their descriptors with the guidance of the matching decision.

User Buffer Registration

To send data from and receive data into user buffer directly, the user buffers need to be registered. Given a non-contiguous datatype we can register each contiguous block one by one. We could also register the whole region which covers all blocks and gaps between blocks. Both attempts have their drawbacks [66]. In [66], *Optimistic Group Registration(OGR)* has been proposed to make a trade off between the number of registration and deregistration operations and the total size of registered space to achieve efficient memory registration on datatype message buffers.

8.2 Performance Evaluation

In this section we evaluate and compare the performance of our SGRS scheme with the Multi-W zero-copy scheme and the Generic scheme in MVAPICH. We perform

latency, bandwidth, bi-directional bandwidth and CPU overhead tests using a vector datatype to demonstrate the effectiveness of our scheme. Then we show the potential benefits that can be observed for collective communication such as MPI_Alltoall that are built on top of point-to-point communication. Further, we investigate the impact of layout caching for our design. Another aspect of our evaluation is the impact of our zero-copy scheme on different platforms. The evaluation has been done on two different platforms. one platform based on PCI-X and the other based on PCI-Express.

Experimental Testbed

For our experiments we used two clusters whose descriptions are given below.

- PCI-X based cluster: A cluster of 8 nodes, each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, 2GB main memory, 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.

- PCI-Express based cluster: A cluster of 4 nodes, each with dual Intel Xeon 3.4 GHz processors and 512MB DDR main memory. The nodes support 8x PCI-Express and connected to Mellanox InfiniHost MT23108 DualPort 4x HCAs. The nodes are connected using an InfiniScale switch. The kernel version used is Linux 2.4.21-15.EL.

Microbenchmarks

In these benchmarks, increasing number of columns in a two dimensional M*4096 integer array are transferred between two processes. These columns can be represented by a vector datatype. We set up two cases for the number of rows (M) in this array: case 1 with 64 rows and case 2 with 128 rows. Basically case 1 has a 'degree of non-contiguity' 64 and case 2 has a 'degree of non-contiguity' 128. The number of columns is varied from 4 to 2048, the corresponding message size varies from 2 KBytes to 512 KBytes. The latency, bandwidth and bidirectional bandwidth experiments use this setup.

Latency

The latency test is a ping-pong latency test with the vector datatype described above. The PCI-X latency results for cases 1 and 2 are shown in Figure 9.4 and Figure 9.6. For each case we compare the two zero-copy schemes (SGRS and Multi-W) and the Generic copy based approach. We also compare it with the latency of the contiguous transfer which serves as the lower bound. When the message size is small, the Generic scheme does better than the zero-copy schemes. This is because, for this range, the copy cost is not substantial whereas the overhead associated with posting the descriptors for the non-contiguous segments dominate. Beyond a cut-off point, 32K in case of SGRS scheme, the zero-copy schemes start outperforming the Generic scheme by a significant margin. Beyond the cut-off point the SGRS scheme does better than the Multi-W. This difference also increases when the degree of non-contiguity increases because Multi-W scheme needs to post a descriptor for each segment individually. We observe that the SGRS scheme reduces the latency by up

to 61% compared to that of the Multi-W scheme. On PCI-Express platforms almost similar trend can be observed for latency for the two cases as seen in Figure 9.5 and Figure 9.7 except that the gap between the SGRS scheme and the Multi-W scheme widens. SGRS scheme reduces the latency by up to 69% compared to that of the Multi-W scheme. Also on the PCI-Express platform the cut off point beyond which the zero-copy scheme performs better is lowered.

Bandwidth

The bandwidth experiment uses the standard bandwidth test except that the datatype is a vector datatype described above. The PCI-X bandwidth results for cases 1 and 2 are shown in Figure 8.17 and Figure 8.19. The improvement factor over the Multi-W scheme varies from 1.12 to 4.0. It can also be observed that when the degree of non-contiguity is large, the improvement of the SGRS scheme over the Multi-W scheme is higher. This is because the improved network utilization in the SGRS scheme is more significant when there are more non-contiguous blocks of small size. When the block size (the size of non-contiguous segment) is large enough, RDMA operations on each block can achieve good network utilization as well and both schemes perform comparably. For large messages our scheme is able to achieve a bandwidth close to that of the peak contiguous bandwidth. This is due to the fact that the large size of messages assisted by the zero-copy mechanism is able to completely saturate the network which is desirable. On PCI-Express platforms, Figure 8.18 and Figure 8.20 show the bandwidth comparison. The trends seen on PCI-X platform are further magnified in the context of PCI-Express. SGRS scheme performs considerably better than the Multi-W scheme and this performance gap is more prominent in PCI-Express as compared to PCI-X. The improvement factor over Multi-W is upto 7.2 on

PCI-Express platforms. Further both the zero-copy approaches show improvement in bandwidth on PCI-Express platform as compared to PCI-X. The Generic copy based scheme does not show any significant improvement across the two platforms. This can be attributed to the fact that the memory bandwidth on the PCI-Express platform is similar to that of the PCI-X platform, and since the Generic scheme is based on copy, and the memory bandwidth is the bottleneck on our PCI-Express platform, the Generic scheme is not able to leverage the improvement in the network bandwidth.

Bidirectional Bandwidth

The memory bandwidth limitation of copy based schemes can have serious impact when we take a look at the bidirectional bandwidth. In a bidirectional bandwidth test, the non-contiguous data flow takes place simultaneously in both the directions. On a PCI-X platform the bidirectional bandwidth attains a peak of 941MB/s for contiguous data. The SGRS scheme and the Multi-W scheme are able to take advantage of this improvement in the bandwidth whereas the copy based Generic scheme saturates around 548MB/s because the bottleneck is the memory copy. The results are shown in Figure 8.21 and Figure 8.2. Compared to the Multi-W scheme, the SGRS scheme does consistently better and is able to achieve a peak bandwidth of 910MB/s for 512K message. This behaviour stands out further on PCI-Express platform which can achieve a peak bidirectional bandwidth of upto 1920MB/s almost double that of uni-directional bandwidth. The PCI-Express bidirectional bandwidth results are shown in Figure 8.2 and Figure 8.2. The zero-copy based schemes can directly leverage this improvement in the network bandwidth and can achieve a bidirectional bandwidth of 1876MB/s close to that of peak contiguous bidirectional bandwidth whereas there is very little improvement for the copy based scheme. Further compared to the Multi-W

scheme, the SGRS performs significantly better and shows an improvement of up to 3 times.

The new and emerging trends in memory technology like DDR2, QDR, etc. could significantly relocate the bottlenecks in the system, presenting new interesting scenarios for further investigations.

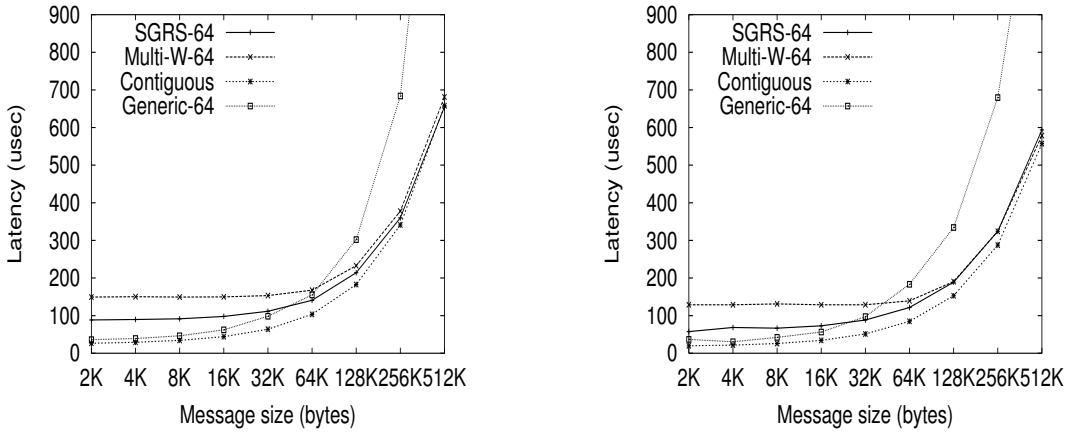

Figure 8.5: MPI Level Vector Latency 64 blocks a)PCI-X and b)PCI-Express

Performance of MPI_Alltoall

Collective datatype communication can benefit from high performance point-to-point datatype communication provided in our implementation. We designed a test to evaluate MPI_Alltoall performance with derived datatypes. We use the same vector datatype we had used for our earlier evaluation.

Figure 8.11a shows the MPI_Alltoall latency performance of the various schemes on 8 nodes for the PCI-X platform. We study the Alltoall latency over the message range 4K-512K. We ran these experiments for two different numbers of blocks: 64 and

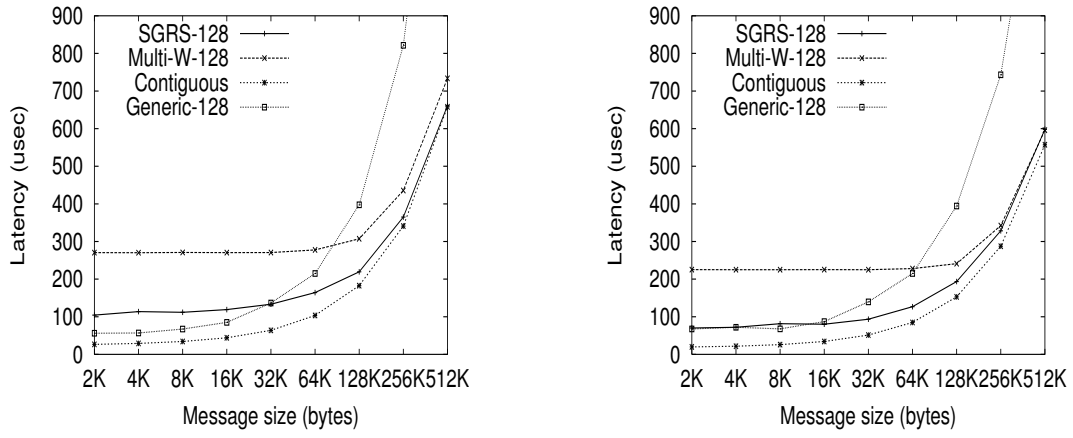

Figure 8.6: MPI Level Vector Latency 128 blocks a)PCI-X and b)PCI-Express

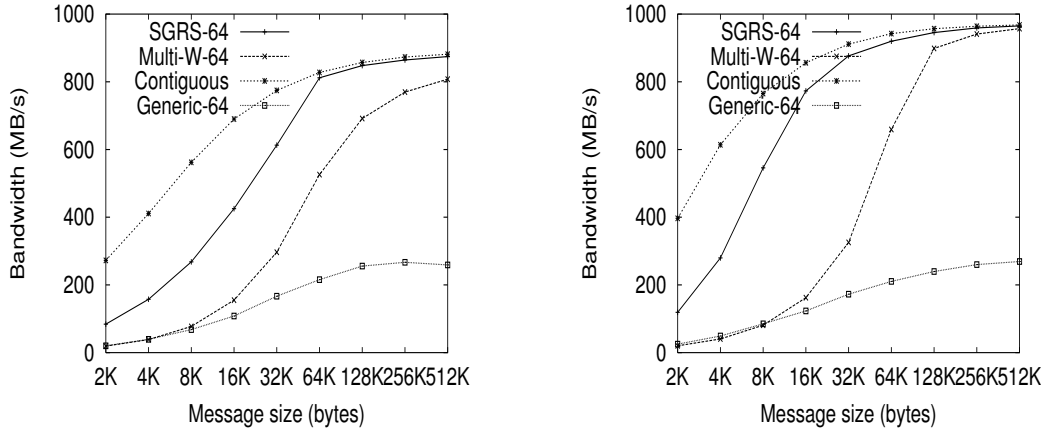

Figure 8.7: MPI Level Vector Bandwidth 64 blocks a)PCI-X and b)PCI-Express

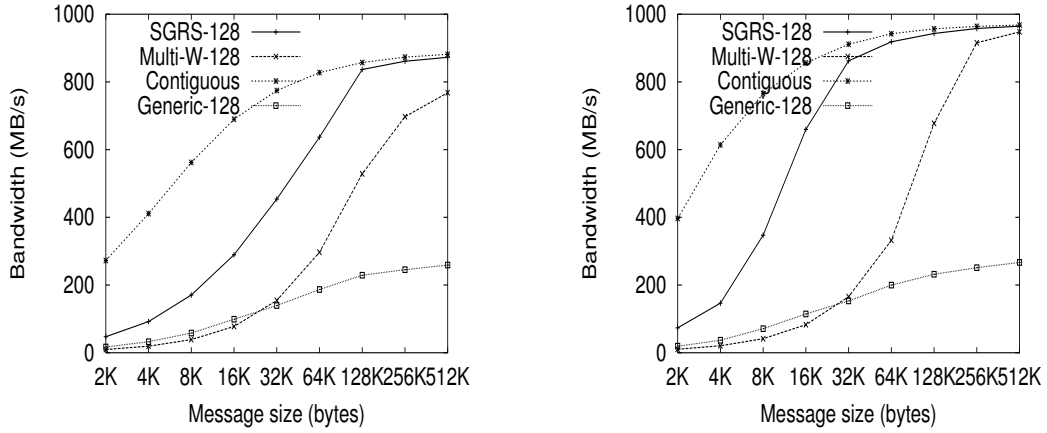

Figure 8.8: MPI Level Vector Bandwidth 128 blocks a)PCI-X and PCI-Express

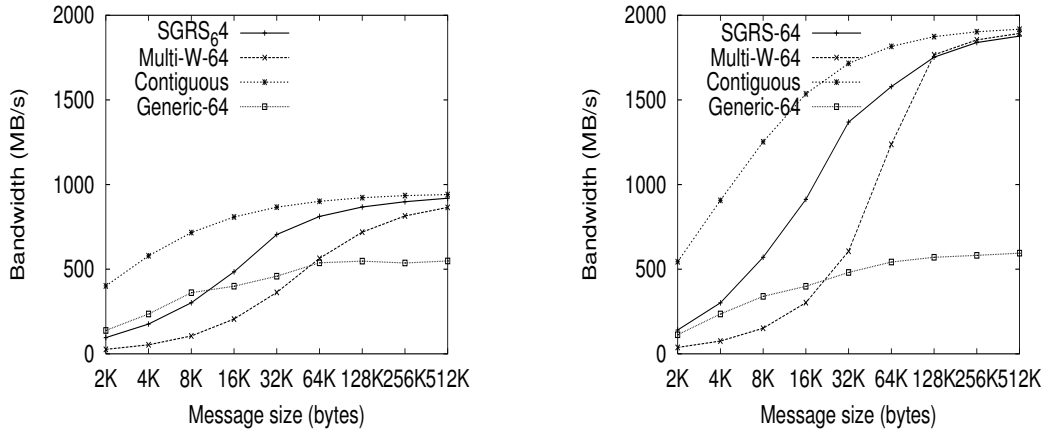

Figure 8.9: MPI Level Vector Bi-directional Bandwidth 64 blocks a)PCI-X and b)PCI-Express

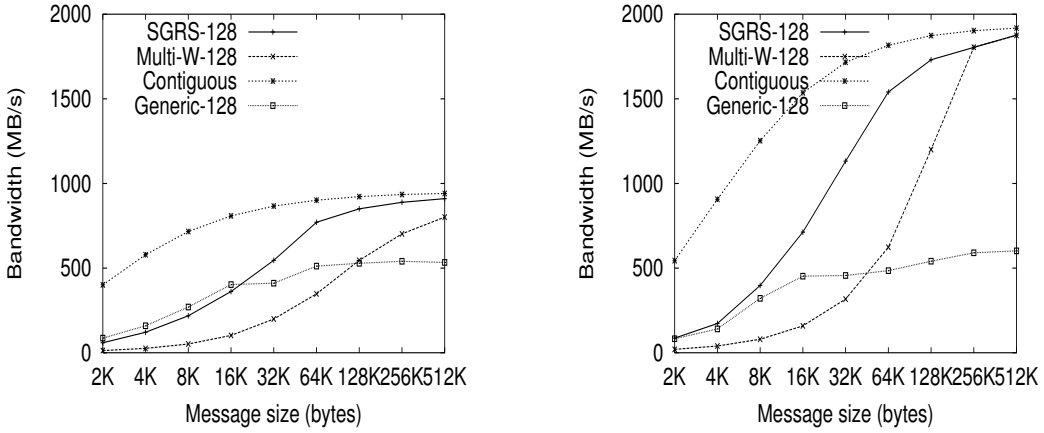

Figure 8.10: MPI Level Vector Bi-directional Bandwidth 128 blocks a)PCI-X and b)PCI-Express

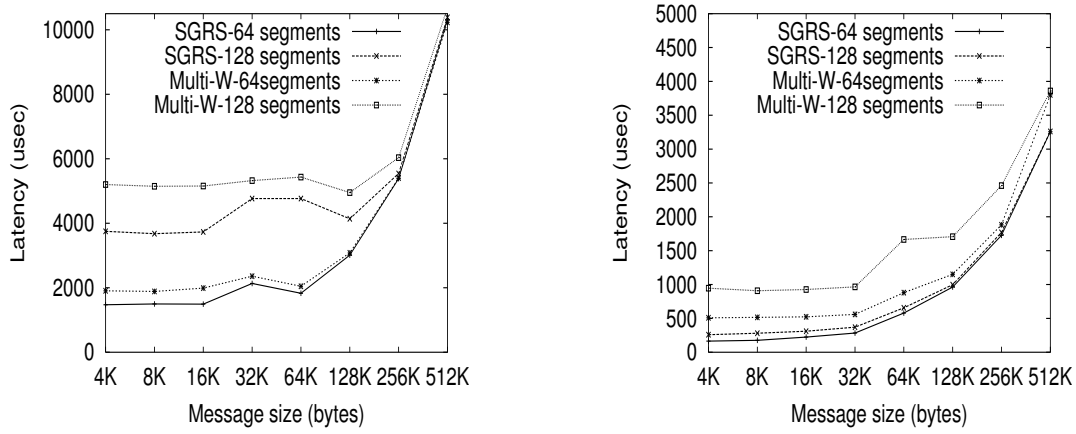

Figure 8.11: MPI_Alltoall Vector Latency a)PCI-X and b)PCI-Express

128. We observe that the SGRS scheme outperforms the Multi-W scheme consistently. The gap widens as the number of blocks increases. This is because the startup costs in the Multi-W scheme increase with the increase of the number of blocks. In addition, given a message size, the network utilization decreases with the increase of the number of blocks in the Multi-W scheme.

The MPI_Alltoall latency performance for PCI-Express platform was evaluated on 4 nodes. The results are shown in Figure 8.11b. The SGRS scheme performs better than the Multi-W scheme and this performance difference is higher on the PCI-Express platform as comapared to PCI-X platform.

CPU overhead evaluation

In addition to the latency and bandwidth, the host CPU usage for the message transfer is also a relevant metric, because it indirectly gives an estimate of CPU availability for the application progress. In this section we measure the CPU overhead involved for the two schemes. These tests were conducted on the PCI-X platform. Figures 8.12 and 8.13 compare the CPU overheads associated at the sender side and receiver side, respectively. The SGRS scheme has lower CPU involvement on the sender side as compared to Multi-W scheme. However on the receiver side the SGRS scheme has an additional overhead as compared to practically close to zero overhead in case of Multi-W scheme.

Impact of Layout Caching

In both the Multi-W and SGRS schemes, the layout has to be exchanged between the sender and receiver before data communication. In this test, we studied the overhead of transferring the layout information. We consider a synthetic benchmark

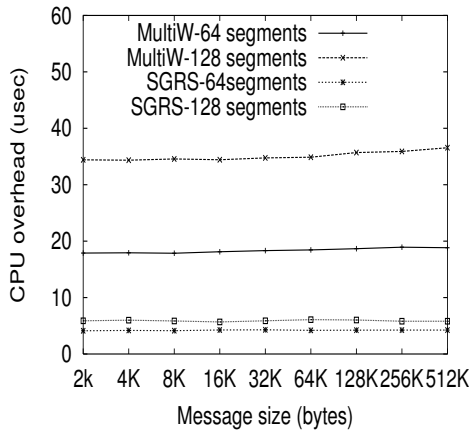

Figure 8.12: Sender side CPU overhead

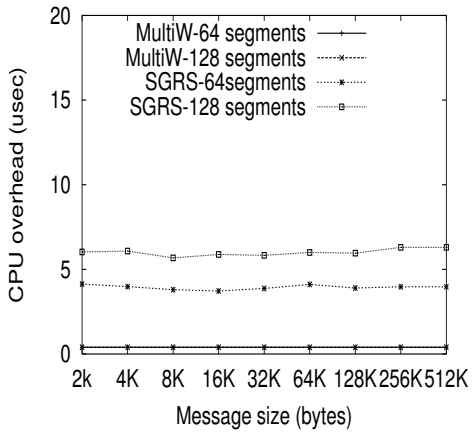

Figure 8.13: Receiver side CPU overhead

where this effect might be prominent. In our benchmark, we need to transfer the two leading diagonals of a square matrix between two processes. These diagonal elements are actually small blocks rather than single elements. Hence, the layout information is complex and we need considerable layout size to describe it. As the size of the matrix increases, the number of non-contiguous blocks correspondingly increases as well as the layout description.

Figure 8.14 shows the percentage of overhead that is incurred in transferring this layout information when there is no layout cache as compared with the case that has a layout cache. For smaller message sizes, we can see a benefit of 10 percent and this keeps diminishing as the message size increases. Another aspect here is that even though for small messages the layout size is comparable with message size, since the layout is transferred in a contiguous manner, it takes a lesser fraction of time to transfer this as compared to the non-contiguous message of comparable size. Since

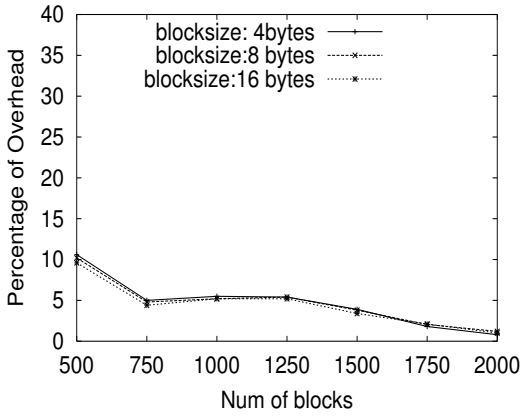

Figure 8.14: Overhead of Transferring Layout Information

the cost associated in maintaining this cache is virtually zero, for message sizes in this range we can benefit from layout caching.

8.3 Non-contiguous One-sided Data-transfer

In this section we address how to handle non-contiguous data transfer efficiently in the context of one-sided communication. In one-sided communication, both the local and remote locations are specified on the sender or origin side. The approaches described in the previous section can also be used for one-sided communication. This work was done in the context of ARMCI which is a one-sided communication library described in section 2.5. We use a helper thread based design which involves limited remote host involvement to provide this support. We intend to extend this design for MPI-2 one-sided communication.

In the following sections we describe a basic approach and our proposed zero-copy approach to handle non-contiguous data transfer in ARMCI library.

8.3.1 Host-Based Buffered Approach

A simple way of performing non-contiguous transfers is to maintain a contiguous buffer on both the local and the remote side and move data using this contiguous buffer. This approach requires heavy involvement on both the local and remote sides in moving the data between the buffer and the noncontiguous source or destination. An enhancement to this approach is to divide the data into chunks and pipeline the memory copy and nonblocking communication so that they overlap. Based on the message size, the message transmission/reception can be broken into smaller requests. A copy of one part of the request can be overlapped with the transmission of another piece. Fig. 8.15 shows the steps involved in a host-based buffered protocol.

Another approach that can be used here is to do multiple contiguous transfers for each contiguous chunk. We refer to this approach as Multiple_Zero_Copy approach. This approach is zero-copy but may require the initiator of the request to spend some time in processing the multiple contiguous requests it has to initiate for every noncontiguous request. In addition, handling flow control issues like the number of outstanding requests allowed might adversely affect performance. We introduced a host-assisted zero-copy method to address the problems inherent in both the approaches described above.

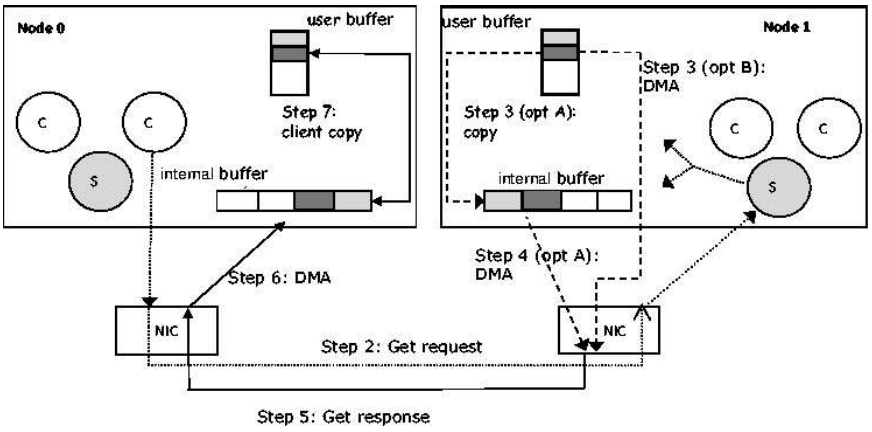

Figure 8.15: Host Based Buffered Approach

8.3.2 Host-Assisted Zero-Copy RMA

To leverage the advantages of the host-assisted zero-copy approach in Mellanox VAPI, memory on both sides must be registered. The user is not expected to either explicitly register memory or keep track of this information. Instead we maintain and parse a high-granularity global memory information table to determine if the memory

on both sides is registered. The host-assisted approach requires partial involvement of a remote host to complete operations. We refer to the representative on the remote side that assists in the completion of the operation as a "helper" thread. The helper thread initiates an operation and hence requires minimal remote-side CPU involvement. This is very similar to the ARMCI data server thread [47, 49] and the dispatcher thread in the IBM LAPI [53]. The significant difference is that the helper thread does not copy any data and does not wait on an operation it issued to complete. With this helper thread as an assistant to complete the operation on the remote side, we describe the implementation details of contiguous and noncontiguous one-sided Get and Put operations. We demonstrate the benefits of this approach by contrasting its performance with the traditional host-based/buffered approach and by showing the performance of these protocols on a few application benchmarks in Section 8.4.

Implementation of Get Operation for Noncontiguous Data

Because a noncontiguous data transfer would involve transfer of multiple segments of data, our strategy is to use the scatter/gather message passing feature provided by IBA to achieve the zero-copy transfer. Using that feature, we can send /receive multiple data segments as a single message by posting a single scatter/gather descriptor. The two types of scatter/gather message-passing operations defined in IBA VAPI are 1) Gather-Send (which requires the noncontiguous data being sent to be represented as a Gather-Send descriptor) and 2) Scatter-Receive (which requires the noncontiguous destination for the receive to be specified in a Scatter-Receive descriptor format). In a host-assisted zero-copy Put, the source sends a request to the remote side. The helper thread processes the request, converts the vector/stride information in the

request into a VAPI Receive-Scatter descriptor, posts the descriptor, and sends an acknowledgment to the requesting process, indicating that it has posted the necessary receive descriptor. On receiving this acknowledgment, the source process posts a Gather-Send from the VAPI Gather-Send descriptor it created while waiting for an acknowledgment from the helper thread. This directly delivers the data to the destination memory without the overhead of any intermediate copies. Although the explicit acknowledgment might seem like an overhead for large messages, when the copying cost starts to dominate, this approach performs better. It should be enabled only for multidimensional Put operations when the first stride or the size of each contiguous segment is large. For a host-assisted zero-copy Get shown in Fig. 8.16, the source node posts a Scatter-Receive descriptor to receive the vector/strided data and then sends a request to the remote host with the remote stride/vector information. The helper thread on the remote host receives the request and then posts a corresponding VAPI Gather-Send by converting the stride/vector information in the request message into a VAPI Gather-Send descriptor. The implementation of this protocol prompted us to address a number of design issues.

Limit on Scatter/Gather Entries per Descriptor

The strided put/get operations can be used to transfer sections of multidimensional arrays. Each dimension of the array can support any number of data segments. However, the IBA implementation puts an upper limit of 60 on the number of scatter/gather entries that can be allowed per Scatter-Receive or Gather-Send descriptor. Hence, for large messages, the maximum scatter/gather entry limit requires us to extend the above approach. Because we can have only 60 scatter/gather entries in a descriptor, our solution is to break our message into chunks of up to 60 data segments

and post a gather send/scatter receive for each one of them. Posting a send/receive is a nonblocking operation in IBA and takes only a very short time (a microsecond on Itanium 1GHz), so the overhead in posting multiple gather descriptors is not significant. In the case of Strided Get, the client posts multiple scatter receives and then sends the request. At the remote side, the helper thread processes the request and posts multiple gather sends. A similar approach has been followed for implementing the noncontiguous puts.

Resource Allocation

At the client level, memory needs to be allocated and maintained to create a scatter/gather descriptor from a strided/vector request. Unlike VIA, VAPI copies the posted descriptor on to the NIC and hence does not require us to keep the descriptor until the request has been completed. At the NIC level, the number of scatter/gather entries must be decided at the initialization phase. The larger the scatter gather list, the larger the amount of memory allocated per descriptor on the NIC. To investigate the effect of this on the performance of the operation, we conducted experiments to measure the change in latency with increasing number of scatter/gather entries. We determined that the overhead for having 60 scatter gather entries in a descriptor instead of 1 is not significant (less than 1 micro sec) and hence we could afford to set the scatter/gather limit to the maximum allowed value of 60.

8.4 Performance Evaluation

We compared the performance of the different methods described above not just to contrast the host-assisted zero copy with the other implementations but also to

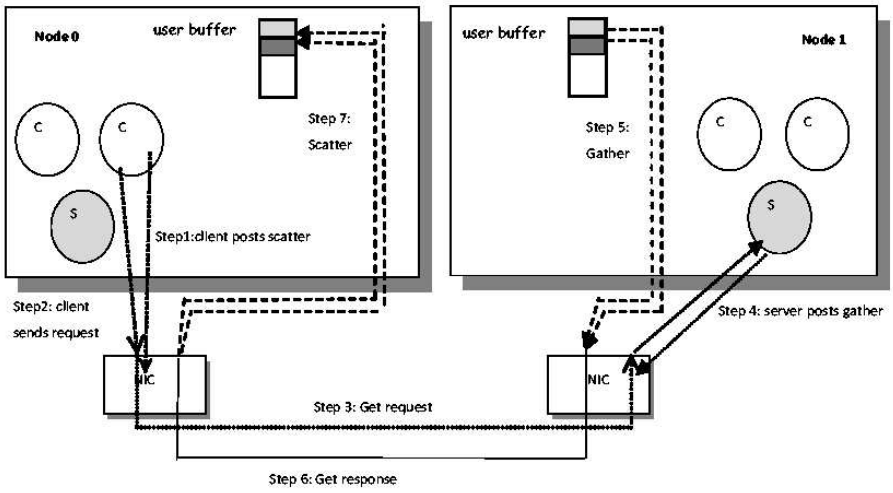

Figure 8.16: Host Assisted Zero-copy Approach

show the importance of using multiple protocols in achieving a sustained good performance. Fig. 8.17 shows the performance of noncontiguous ARMCI operations. It compares the performance of host-based/buffered get and host-assisted zero-copy get operations.Zero-Copy 2D get in Fig. 8.17 and Fig. 8.18 represents the approach discussed earlier in this section where a noncontiguous Get operation is implemented on top of multiple contiguous RDMA Get operations, one for each contiguous segment. For this test, ARMCI 2D data is represented using the strided data format. It is clear that the host-assisted zero copy implementation performs much better and more significantly so when the first dimension is large. An advantage of using host-assisted zero copy can be determined by measuring the effect on protocol performance when the remote side is doing a CPU-intensive operation. Unlike the zero-copy approach, host-assisted zero-copy requires some host involvement in initiating data transfer. This is more representative of the impact these protocols may have on an application

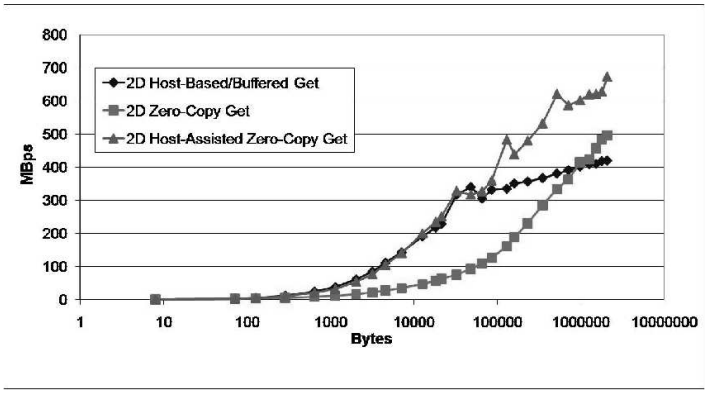

Figure 8.17: Bandwidth Comparison with Remote Side Idle

than mere measurement of communication bandwidth/latency. Fig 8.18 shows the performance difference between the buffered and host-assisted zero-copy protocols when the remote side is doing a CPU-intensive operation. In comparison to Fig. 8.17, it is very clear that the performance of the host-assisted zero-copy protocol has not been affected at all by the CPU-intensive operation on the other side while the performance of the buffered Get protocol dropped very significantly. This clearly shows the very low overhead this protocol imposes on the remote-side CPU.

Overlap Measurements

Another significant advantage of this protocol is the amount of overlap it can provide in nonblocking operations. Because the implementation does not involve any data movement in call initiation or call completion, the amount of overlap possible is much higher than that for the other protocols. This can be clearly seen in Fig 8.19,

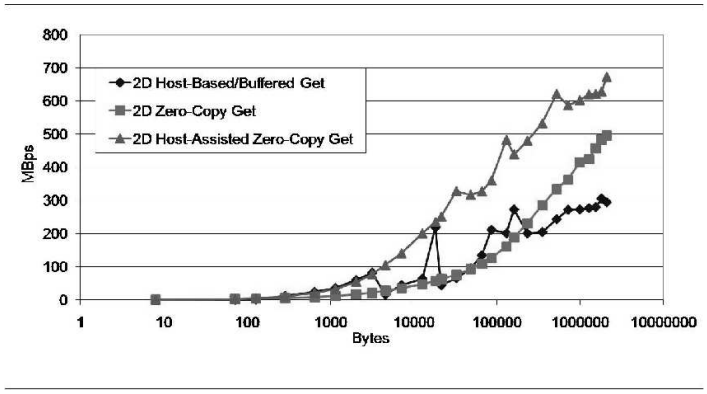

Figure 8.18: Bandwidth Comaprison with Remote Side Busy

which compares the amount of overlap attainable with host-based/buffered and host-assisted protocols for a noncontiguous data transfer for various square noncontiguous chunks of data.

Matrix Multiplication

The bare microbenchmark performance numbers for RMA operations often do not give the actual impact of the protocol used to implement the one-sided operation on an application. A significant issue that comes to light in actual application performance in the case of one-sided operations is the ability of the operation to make progress with minimal to no remote host involvement. SUMMA is a highly efficient, scalable implementation of common matrix multiplication algorithm proposed by van de Geijn and Watts [27]. For the RMA version, we used the algorithm implemented using ARMCI RMA in Global Arrays. The matrix in the Global Arrays implementation of ARMCI

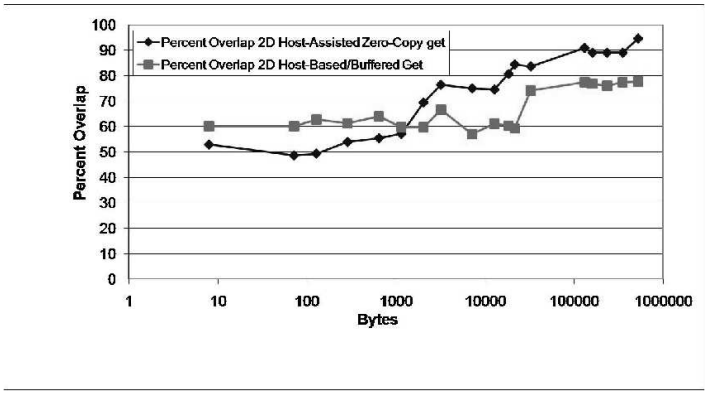

Figure 8.19: Overlap Percentage

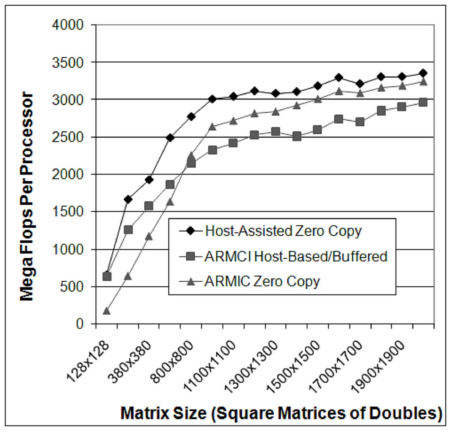

Figure 8.20: Performance of Matrix Multiplication for Square Matrices

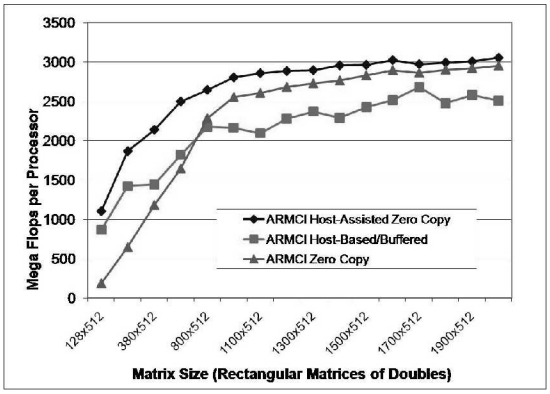

Figure 8.21: Performance of Matrix Multiplication for Rectangular Matrices

is decomposed into blocks and distributed among processors with a two-dimensional block distribution. Each submatrix is divided into chunks. Overlapping is achieved by issuing a call to get a chunk of data while computing the previously received chunk. The minimum chunk size was 128 for all runs, which was determined empirically. The maximum chunk size was determined dynamically, depending on memory availability and the number of processors. Experiments with matrix multiplication were run by varying the matrix size and the number of processors. The three lines labeled in both the graphs in Fig. 8.20 represent three different approaches to implement multi-dimensional RMA in ARMCI. The host-assisted zero-copy approach was introduced in Section 8.3.2. The host-based/buffered approach and zero-copy approaches were discussed at the beginning of Section 8.3.1. The host-based/buffered approach involves two copies, one on each side; the zero-copy approach involves multiple contiguous sends for each noncontiguous message. The computations were done on four nodes with two processes each. Fig. 8.20 shows the result for square matrices with

sizes varying from 128 to 2000. Fig. 8.21 is for a rectangular matrix where the second dimension is set to 512 and the first dimension varies from 128 to 2000. Our proposed host assisted approach outperformed the other schemes for microbenchmarks as well as application kernels like SUMMA matrix multiplication.

This work described how non-contiguous one-sided communication can be implemented efficiently through the novel host-assisted approach to support the zero-copy communication. In addition, a high degree of overlapping computations and communication was demonstrated. The benchmarks used in the study showed effectiveness of the RMA implementation on InfiniBand and the importance of zero-copy nonblocking protocols for hiding latency in the interprocessor communication.

8.5 Related Work

Many researchers have been working on improving MPI datatype communication. Research in datatype processing system includes [32, 52]. Research in optimizing packing and unpacking procedures includes [18, 32]. The closest work to ours is the work [67] to take advantage of network features to improve noncontiguous data communication. In [67], Wu *et al.* have systematically studied two main types of approach for MPI datatype communication (*Pack/Unpack-based approaches* and *Copy-Reduced approaches*) over InfiniBand. The Multi-W scheme has been proposed to achieve zero-copy datatype communication.

CHAPTER 9

NON-BLOCKING ONE-SIDED PRIMITIVES

As described in earlier sections, a one-sided communication library should provide low latency one-sided operations and good scope for overlap potential. Non-blocking operations are very important to achieve latency hiding and good computation/communication overlap. Nonblocking operations initiate a communication call and then return control to the application. The application writer/user can try to hide the latency of the communication operation by overlapping communication with computation.

There are two important aspects to this issue. The first is the availability of the non-blocking API that can be exposed to the application writers. Secondly, the underlying implementation needs to be non-blocking to achieve this. In this work, shown in the highlighted part of Figure 9.1 of the proposed research framework, we explore techniques and designs to implement these non-blocking primitives in the context of ARMCI which is a one-sided communication library.

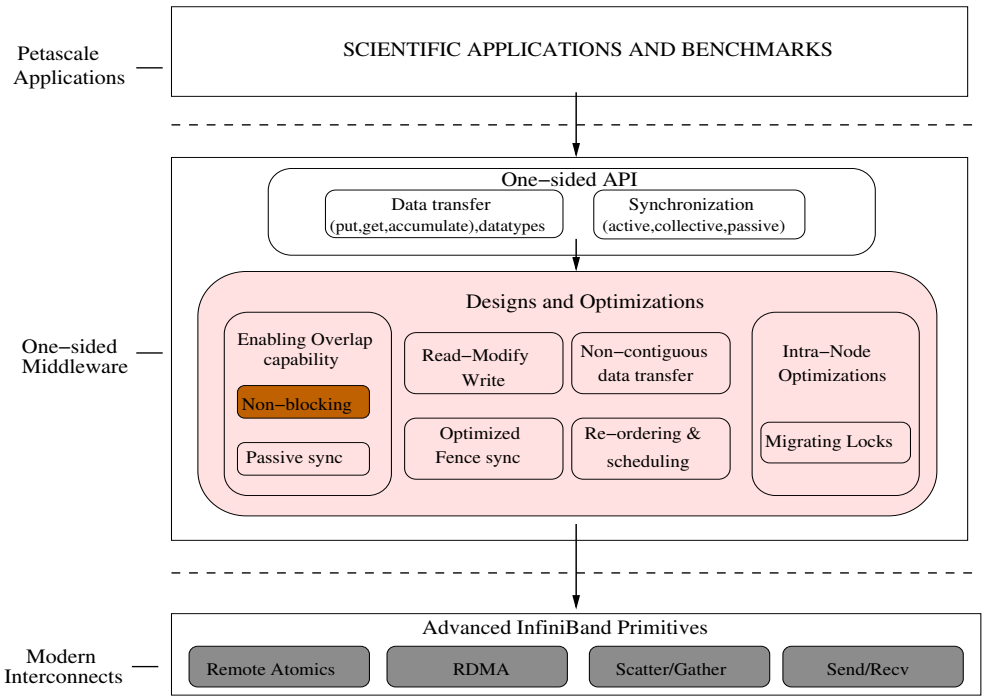

Figure 9.1: Overview

9.1 Efficient Non-blocking Design

Nonblocking operations initiate a communication call and then return control to the application. The user who wishes to exploit nonblocking communication as a technique for latency hiding by overlapping communication with computation implicitly assumes that progress in communication can be made in a purely computational phase of the program execution when no communication calls are made. Unfortunately, that assumption is often not satisfied in practice, the availability of nonblocking API does not guarantee that overlapping communication with computation is always possible [65].

Since the RMA or one-sided model is simpler than two-sided message passing model (e.g., does not involve message tag matching or dealing with early arrival of messages), in principle more opportunities for overlapping communication with computation are available. However, these opportunities are not automatically exploited by deriving implementations of nonblocking APIs from their blocking counterparts. For example, the communication protocols used to optimize blocking transfers of data from non-registered memory by pipelined copy and network communication through a set of registered memory buffers [49] can achieve very good performance by tuning the message fragmentation in the pipeline [64]. However, the memory copy requires the active host CPU involvement and therefore reduces the potential for effective overlapping communication with computation. To increase the overlap, we expanded the use of direct(zero-copy) protocols on networks that require memory registration, such as Myrinet.

In ARMCI, a return from a nonblocking operation call indicates a mere initiation of the data transfer process, and the operation can be completed locally by making a call to the wait routine. Waiting on a nonblocking put or an accumulate operation ensures that data was injected into the network and the user buffer can be now be reused. Completing a get operation ensures that data has arrived into the user memory and is ready for use. A wait operation ensures only local completion. The library imposes a limit on the number of outstanding requests allowed (if necessary, it can transparently complete an old request and free up the resources for a new request). For performance reasons [12], ARMCI supports only a weak consistency for operations targeting remote memory. Unlike their blocking counterparts, the nonblocking operations are not ordered with respect to the destination. Performance

is one reason; the other is that by ensuring ordering, we incur additional and possibly unnecessary overhead on applications that do not require ordered operations. When necessary, ordering can be done by calling a fence operation. The fence operation is provided to the user to confirm remote completion if needed.

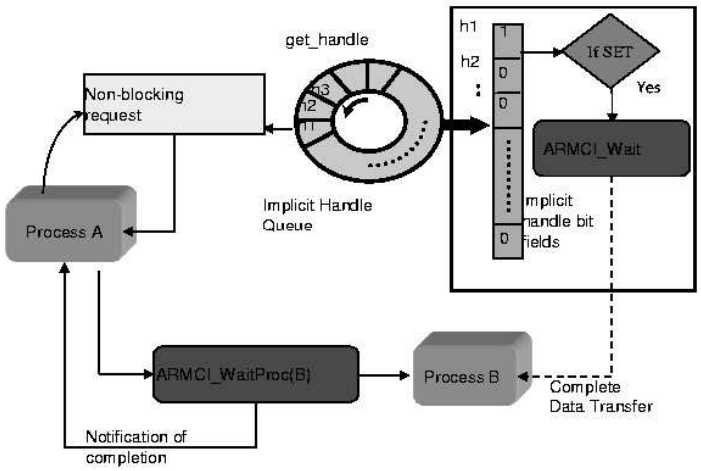

Figure 9.2: Non-blocking transfer with implicit handle

Request Handle

The request handle structure is central to the APIs associated with the latency hiding mechanisms in ARMCI. This opaque object is stored in the application memory and is used to 1) assign a unique identity to a nonblocking RMA operation, 2) facilitate aggregation of multiple operations, and 3) optionally store certain control information. Before the handle is used, it must be initialized with the ARMCI_INIT_HANDLE macro and can be reused after the associated nonblocking operation completes. The

user passes a reference to a request handle structure. As a convenience to the user, a NULL value for the handle address can be specified. The library keeps track of these so-called "implicit handle requests" and assigns a handle to them from an internal pool of handles. This type of requests can be completed using either the wait operation associated with a particular remote processor (see Fig. 9.2) or another wait operation to complete all pending implicit handle requests.

9.2 Implicit and Explicit Aggregation

Aggregation of requests is another mechanism for improving latency tolerance. Multiple nonblocking data transfer (put/get) requests can be aggregated into a single data transfer operation in order to improve the data transfer rate. Especially if there are multiple data transfer requests of small message sizes, aggregating those requests into a single large request reduces the latency, thus improving performance. This technique is unique in its ability to sustain high bandwidth utilization and enables high throughput. Each of these requests can be of a different size and independent of data type. The aggregate data transfer operation is independent also of the type of put/get operation; that is, it can be a combination of regular, strided, or vector put/get operations. There are two types of aggregation available: 1) explicit aggregation, where the multiple requests are combined by the user through the use of the strided or generalized I/O vector data descriptor, and 2) implicit aggregation, where the combining of individual requests is performed by ARMCI. The implicit aggregation involves the nonblocking request handle that is marked as "aggregate handle" using the ARMCI_SET_AGGREGATE_HANDLE macro. Users can rely on a single aggregate handle to represent multiple requests. Any number of operations

to/from the same processor can use the same aggregate handle. A wait on such a handle completes all the aggregated requests. For multiple small sends, aggregating is usually much faster and gives better performance. Fig. 9.3 illustrates the aggregate data transfer. It shows that the descriptors of multiple put requests are stored in an aggregate buffer and, once the wait call is issued, the data transfer is completed.

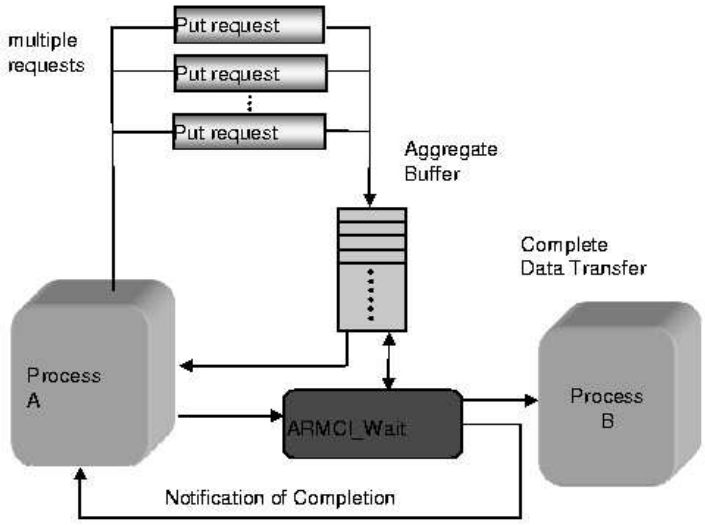

Figure 9.3: Implicit Aggregate Data Transfer

Design and Implementation Approach

Designing a portable RMA communication layer involves addressing multiple issues: 1) the functionality must be implementable across a wide variety of platforms; 2) performance advantages of the native communication protocols must be exploited; 3) opportunities for overlapping communication and computations should be provided;

and 4) as much of the code as possible must be shared to minimize the maintenance efforts across different platforms. On networks like the IBM SP interconnect and Quadrics, the underlying RMA layer provides most of the required capabilities. Hence, on these systems, most of the nonblocking calls can be implemented as thin wrappers to the native protocols. We are referring to these protocols as direct. In the case of some networks, direct protocols are zero-copy (GM, VIA, Quadrics Elan), but others where the native communication interface involves copying the data (IBM LAPI) internally are not. Some networks like GM, VIA, and Infiniband require data to be transmitted from/to special memory. This can be accomplished either by 1) copying the data into a set of special registered/pinned buffers for transmission; 2) allocating registered memory for the user; or 3) by on-demand registration of the user's memory. ARMCI uses all three schemes, depending on the platform, operation type, or size of the data transfer. Protocols that use memory copy scheme are referred to as buffered. Although the goal is to generalize most of the design, doing so should not adversely affect the performance in cases where an underling network provides direct support.

Multiple requirements can be satisfied by a buffer management layer. First, on networks that allow data transfers between registered buffers, the data can be copied in, sent, received, and copied out from the internal set of buffers allocated in registered memory. In this manner, data can be transferred between nonregistered memory locations. Note that on-demand memory registration of user buffers might not always be available or can be very costly (e.g. GM) [49, 14]. Second, buffers are useful for packing/unpacking noncontiguous data transfers when the underlying network has support only for contiguous data transfers (for example, GM) [49]. One of the

design goals is to make most of the handle management code and buffer management code platform-independent, thus making the architecture portable while avoid the unnecessary overhead. This is accomplished by switching to a direct protocol when possible at the very beginning of the request processing. Interaction between the platform-independent layer and platform/network-specific layer is only to either inject the data into the network or check for the completion of an operation.

Handle Management

Every nonblocking call is associated with a nonblocking request handle. For explicit handle nonblocking calls and aggregate handle nonblocking calls, this handle is passed by the user as a parameter. An implicit handle call is associated with a handle from a static list of handles, maintained internally. The handle provided by the user is internally mapped to a data structure that in turn carries all the information required to identify and complete, or test completion of a nonblocking operation. Because a common handle is used to represent a request on all platforms, for portability reasons it stores only the most generic information, including unique identifier of the request, the type of operation, and the remote processor number. Other fields include completion information required by the underlying network for request completion.

Communication Buffers

The communication buffer is represented by a data structure that stores information about the associated request. In nonblocking operations, it also carries a unique request identifier for the request.For the buffered implementation of the get operation, it stores the destination address for the data. For strided and vector operations, the destination information is represented by a more complex descriptor of variable

size. The buffer data structure has a fixed space allocated to store destination data descriptors. For a larger descriptor, extra memory is allocated, and the corresponding address is stored in the buffer. That memory is freed when the operation associated with this buffer is completed. The "protocol" field in the buffer structure carries more detailed information. For example, the "protocol" field in the buffer management phase carries the value "sdescr_in_p", which indicates that this buffer is being used for a strided data transfer and the destination data descriptor is in place (sdescr_in_p) inside the buffer data structure. This information is needed to complete a request. ARMCI does not impose a limit on the number of outstanding operations. Hence, when the buffer management layer runs out of buffers, it completes an old request associated with a buffer currently in use to free a buffer. Because a request can be using more than one buffer, freeing a buffer might complete only a part of the request. A communication buffer is also freed as a part of the wait operation on the request using that buffer.

Waiting on a Request

The wait on a request handle completes the request. Whether the request used buffers or not can be determined by looking at the value stored in the bufid field of the request handle. For the direct protocol, the platform-specific layer verifies request completion based on the information it stored in the "Req completion info" field. If buffers were used for the request (buffered protocol or for storing a data descriptor), then the buffer management layer checks to see if the buffers used for this request were completed already as a part of freeing resources. If they have not yet been completed, then the data from the buffer is copied into the appropriate destination based on the destination descriptor information stored in the buffer. To be able to

verify if the data has already arrived in the buffer, the buffer management layer may check for data arrival via the platform-specific layer.

Aggregation

The implicit aggregation of data transfers is implemented using the generalized I/O vector operations available in ARMCI [47]. This interface enables the representation of a data transfer as a combination of multiple sets of equally sized contiguous data segments. When the first call involving aggregate nonblocking handle is executed, the library starts building a vector descriptor stored in one of the preallocated internal buffers. The actual data transfer takes place when the user calls wait operation or the buffer storing the vector descriptor fills up.

Optimizing Overhead and Overlap

The overhead introduced due to the additional processing and resource management incurred by a nonblocking call should be minimized. In our implementation, this goal is achieved in multiple ways: Before returning, all nonblocking operations always initiate data transfer so that the network interface card (NIC) can process a request while the host CPU is available to carry out the computations. When a nonblocking GET operation returns, either the buffered or direct protocols ensure that all the requested data will be received without explicit involvement of the host CPU. In the buffered protocol, the request is broken into pieces that fit the available buffer space. For very large buffered requests, some initial portion of the data might be received before the nonblocking operation returns. The direct protocol is switched to when possible, as described earlier. The platform-specific protocols that involve extensive blocking time are avoided.

9.3 Performance Evaluation

The experiments were run on a Linux cluster with dual 2.4GHz Pentium-4 nodes and Myrinet-2000 (M3F-PCI64C-2 Myrinet interface). Experiments discussed in the current section have been conducted for the nonblocking get operation since they explicitly demonstrate the overhead and overlap factors.

9.3.1 Overhead Test

The first experiment demonstrates the efficiency of the implementation as compared with a base case GM implementation. For this purpose, a nonblocking operation is simulated at the GM level in the following fashion. The client issues a gm_send_with_callback (with the details of the required data) and then polls on a flag set when the data reaches this node. On the other end, the server does a GM_receive, processes the request, and issues the RDMA put operation with the data using the gm_directed_send_with_callback function. The ARMCI layer is actually built on this basic scheme to implement the nonblocking get. This experiment tries to evaluate the efficiency of the implementation. Fig. 9.4 shows the latency at the base GM and ARMCI levels. The timings have been averaged over 1000 iterations. They show that the ARMCI layer adds very little overhead to the base level and thus provides a very efficient interface to the applications.

9.3.2 Overlap Test

The second experiment deals with overlapping communication with computation, and it was performed in the context of ARMCI and MPICH-GM. In the ARMCI

version, the computation is incorporated in the program in the form of a delay. Increasing computation is gradually inserted between the initiating nonblocking get call and the wait completion call. As we keep increasing the computation, at some point the sum of the nonblocking call issue overhead and computation would exceed the idle CPU time, so the total benchmark running time would increase. This point gives us the maximum possible overlap. We performed this experiment on two nodes, with one node issuing the nonblocking get for data located on the other and then waiting for the transfer to be completed in the ARMCI_Wait call. The timings were averaged over 1000 iterations. We have developed versions of this microbenchmark for direct and buffered protocols. We also implemented an MPI version of the above benchmark because our motivation was to compare the overlap in ARMCI and in the MPI nonblocking send/receive operations. In MPI, if the node needs a portion of data from another node, it sends a request and waits on a nonblocking receive for the response. We can overlap the time duration between these two calls with computation. We measured the computation overlap for both the ARMCI and MPI versions of the benchmark, and results are plotted in Fig. 9.5. The percentage overlap is measured as the amount of time of a nonblocking (data transfer) call that can be overlapped with useful computation without increasing the overall benchmark time.

We observe that ARMCI offers a higher level of overlap than MPICH-GM. The buffered protocol is able to achieve about 90% overlap. For large messages, this percentage drops because of time involved in copying to the destination buffer. In the direct protocol, we are able to overlap almost the entire time (greater than 99%). The exception (1%) was the time involved in issuing the nonblocking get. The MPICH-GM version does reasonably well up to message size 16kb. At 16kb and beyond, the

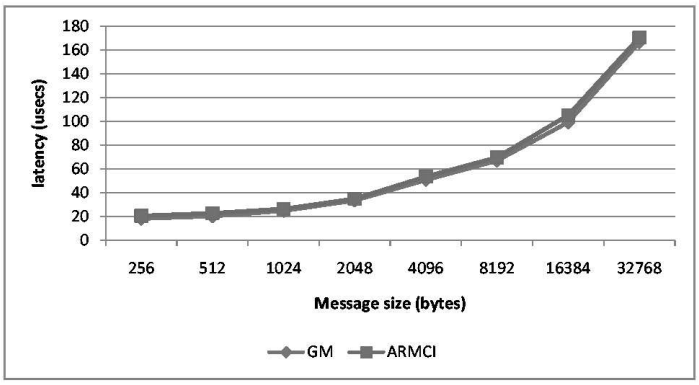

Figure 9.4: Latency of ARMCI Get vs GM Get

MPICH-GM implementation switches to the rendezvous protocol. This has a serious impact on the computation overlap because the handshake involved in the protocol occurs in MPI_Wait. Consequently, the only part that can be overlapped is till the receipt of 'request to send' and not until the actual data transfer is completed.

9.3.3 NAS benchmarks

The Numerical Aerodynamic Simulation (NAS) parallel benchmarks (NPB) are a set of programs designed at NASA. Our starting point was NPB 2.3 [11] implementation written in MPI and distributed by NASA. We modified two of the five NAS kernels, MultiGrid (MG) and Conjugate Gradient (CG), to replace point-to-point blocking and nonblocking message-passing communication calls with first blocking and then nonblocking RMA communication. This is just a mere replacement of the point-to-point message passing communications part of the current message-passing

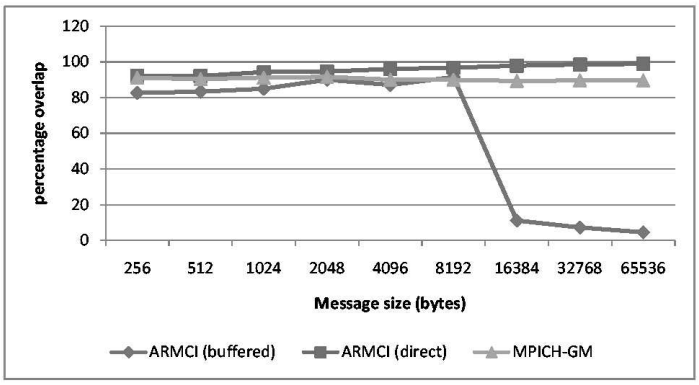

Figure 9.5: Percentage of Computation Overlap

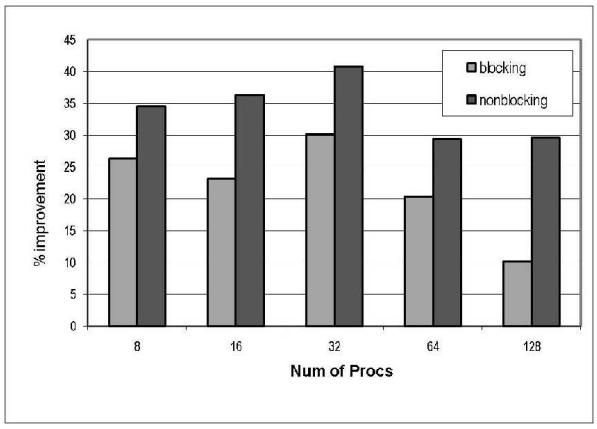

Figure 9.6: Performance Improvement in NAS MG for Class B

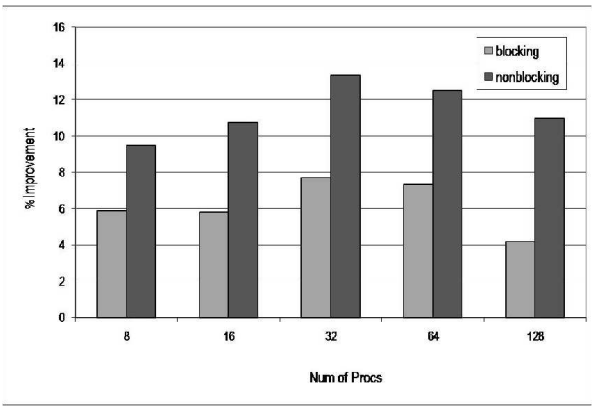

Figure 9.7: Performance Improvement in NAS CG for Class B

version of CG and MG NAS kernels using ARMCI RMA blocking and nonblocking operations [62].

We show the results for NAS MG for class A and B. For Class A, a smaller problem size with the fewest iterations, the ARMCI blocking code outperforms the reference MPI implementation by 7% to 30%. ARMCI nonblocking version achieves an additional improvement of 10% to 23% over the ARMCI blocking implementation and a 28% to 46% improvement over the MPICH-GM implementation. Most of the improvement achieved over the blocking implementation is just by mere issue of the update in the next dimension while working on the current one. For Class B, with the same problem size as class A but more iterations, ARMCI blocking implementation outperforms MPI by 10% to 37% (see Fig. 9.6). The ARMCI nonblocking implementation achieves an additional improvement of 5% to 20% over the blocking version and shows a 30% to 45% improvement over the MPICH-GM implementation. Due to the synchronous nature of data transfers in the CG algorithm, the performance

improvement over MPICH-GM, although consistent is rather limited (see Fig. 9.7). However, the nonblocking RMA offers an additional performance improvement. For example, for 128 processors, it exceeds 10% over MPICH-GM.

CHAPTER 10

SCHEDULING ONE-SIDED OPERATIONS

The MPI-2 semantics does not impose any restrictions on when and in what order the RMA operations should occur within an access epoch. However both the current implementations (Point to Point Based and Direct One Sided) for active synchronization always maintain the order of the RMA operations. This might not always lead to the best or optimum usage of the underlying network capability. In this work, shown in the highlighted part of Figure 10.1 of the proposed research framework, we want to exploit this flexibility to explore different ways to reorder these RMA operations based on the communication pattern to improve the latency, bandwidth and throughput.

Message aggregation can reduce the latency for small RMA opearations because it can potentially reduce the number of messages. The Point to Point Based implementation can give this ability because of its two sided nature. With the Point to Point Based implementation several RMA operations can be reordered and combined/aggregated into a single message and the remote side can receive this combined message and scatter them. Aggregation of a RMA communication operation and a synchronization message is also feasible. Thus the Point to Point Based implementation can be leveraged to improve the performance of small messages.

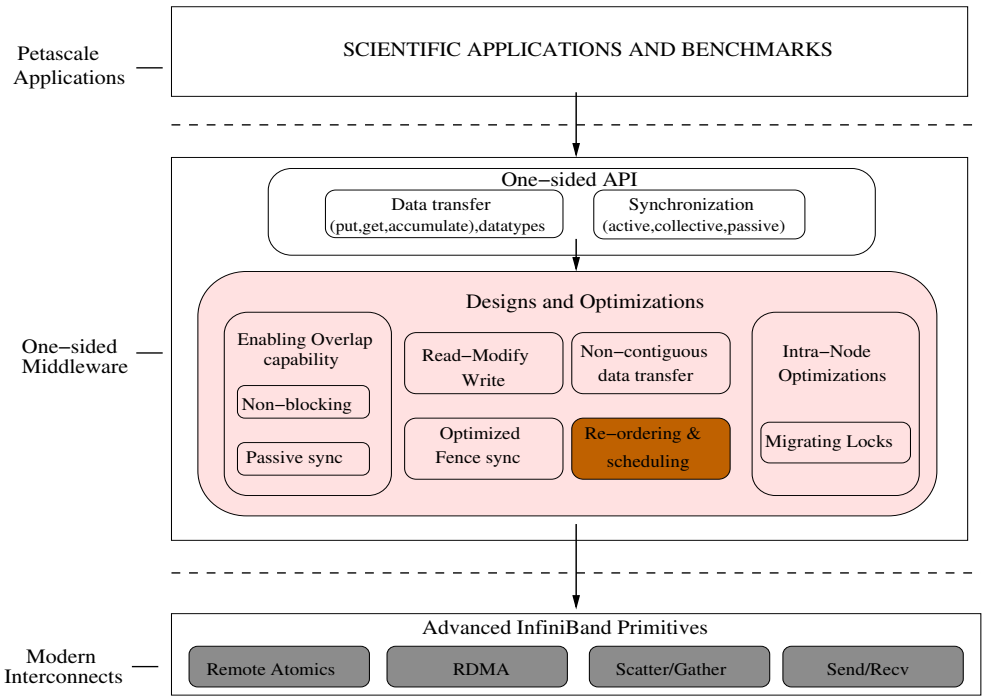

Figure 10.1: Overview

As described above, the MPI-2 semantics potentially allow the implementation to reorder the actual completion of the RMA operations, such as MPI_Put and MPI_Get, issued during a window access epoch. Our main motivation is to utilize this flexibility to schedule these operations so that we can achieve better communication overlap, reduced latency and improved throughput on our InfiniBand implementation.

We propose two possible approaches for scheduling the RMA operations. The reordering approach focuses on reorganizing the MPI_Put and MPI_Get operations issued during a window access epoch to allow more efficient usage of network bandwidth. The aggregation approach tries to combine RMA operations to give better throughput.

10.1 Reordering approach

Since MPI-2 standard allows the actual communication for RMA operations to happen at synchronization time, we can hold all the RMA operations issued during a window access epoch until synchronization time. At this stage, we will have enough information of the communication pattern during this access period. Based on this information, we may re-order the issuing of these RMA operations to utilize the underlying InfiniBand network more efficiently.

10.1.1 Interleaving

The bidirectional bandwidth is always higher than the unidirectional bandwidth. This is because of the full usage of the link bandwidth of both directions. For example, with MVAPICH2 point to point communication, we are able to achieve 874MB/s peak unidirectional bandwidth while we can achieve 934MB/s in bidirectional bandwidth test. (The unit of bandwidth MB/s refers to Million bytes/sec). This trend is more obvious on PCI-Express systems because the bus contention is no longer the bottleneck in this scenario. The peak bandwidth number for unidirectional and bidirectional tests are 964MB/s and 1905MB/s on the PCI-Express system.

However, in a typical one-sided communication scenario, only one direction of the link bandwidth is fully used, since the target side is not explicitly involved in the communication. But this does not mean that we can only stick with the highest possible unidirectional bandwidth provided by the link. For MPI_Put operations, we issue RDMA write operations at VAPI level to push the data out. The actual data flow is from the origin process to the target. But for MPI_Get operation, we issue RDMA read operation at VAPI level to fetch data from the remote side. So the actual

data flow, especially for large size operations, is from the target process to the origin process.

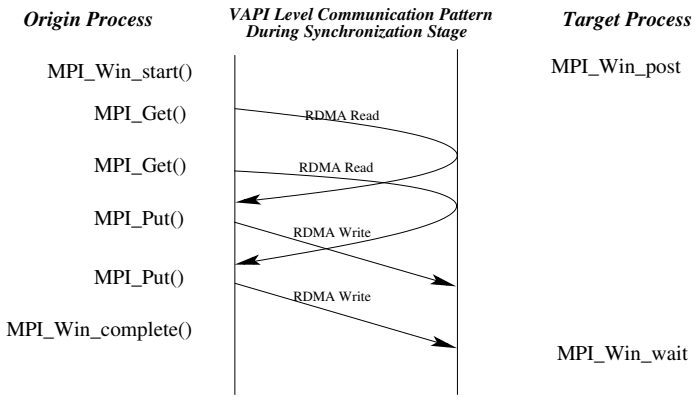

Figure 10.2: Sequential Issue of MPI_Get and MPI_Put

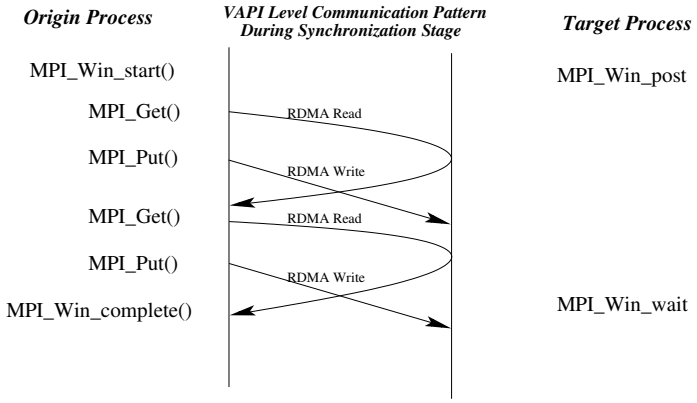

Figure 10.3: Interleaved Issue of MPI_Get and MPI_Put

Let us consider the following one-sided communication patterns. In Fig. 10.2, the origin process issues several MPI_Get operations and then several MPI_Put operations

during a RMA access epoch. In Fig. 10.3, the origin process issues the same number of MPI_Get and MPI_Put operations, but in an interleaved way. As we can observe, the second communication pattern in Fig. 10.3 can use the link bandwidth in a much more efficient way than the first communication pattern.

Though we know that the link bandwidth will be used more efficiently if the issuing of MPI_Put and MPI_Get is interleaved, we can not require the MPI programmer to understand this and always write the optimized program. But since the RMA operation can actually start during synchronization time, we can schedule the operations so that the corresponding VAPI level RDMA read and RDMA write operations are issued in a interleaved manner.

10.1.2 Prioritizing

One of the conclusions of our previous research is that the Direct One Sided implementation offers better latency than Point to Point Based implementation for large RMA operations. But it is still possible to further optimize the Direct One Sided implementation.

During the synchronization stage of direct one-sided implementation, the origin process will issue a RDMA write to set a flag at the target process to indicate the end of the access epoch. Before that, if a MPI_Get operation was issued prior to the synchronization call, we need to wait for local completion of Get to ensure that the data has actually been fetched and ready for use by the end of synchronization phase.

During the access epoch, if the origin process calls several MPI_Put and MPI_Get operations, we want to give priority to MPI_Get operations in order to reduce the time involved in waiting for the local completion. Therefore we give priority to MPI_Get

operations over MPI_Put operations. We first issue RDMA read required by MPI_Get and then issue RDMA write required by MPI_Put. Fig. 10.4 illustrates the potential benefits of our prioritizing scheme. It is to be noted that this prioritizing scheme does not necessarily contradict with the interleaving scheme we proposed in the last section. We can still interleave the operations but we can issue RDMA read operations first.

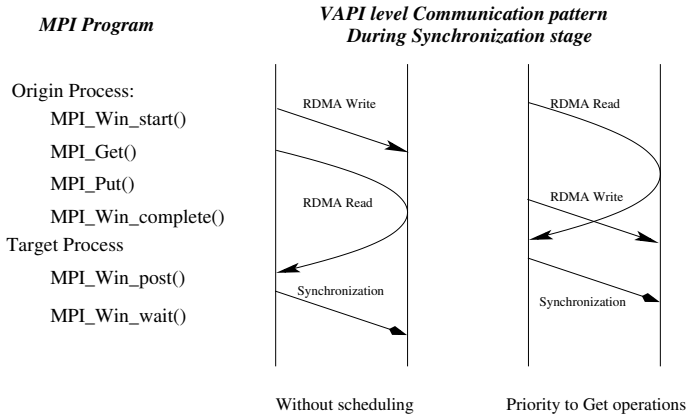

Figure 10.4: Potential Benefit by Giving Priority to MPI_Get

10.2 Aggregation

As described earlier, our goal here is to better utilize the network bandwidth. If we have multiple small RMA messages within an access epoch, the network utilization would be suboptimal. Because, for small messages, the overhead associated with initiation and completion of RMA operations is relatively high. Hence a natural and obvious choice would be to try and see if we can aggregate several of these messages

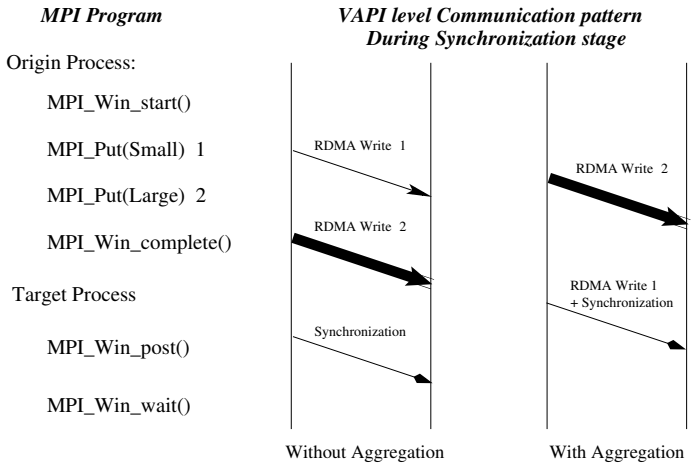

Figure 10.5: Aggregation of RMA Operation and Synchronization

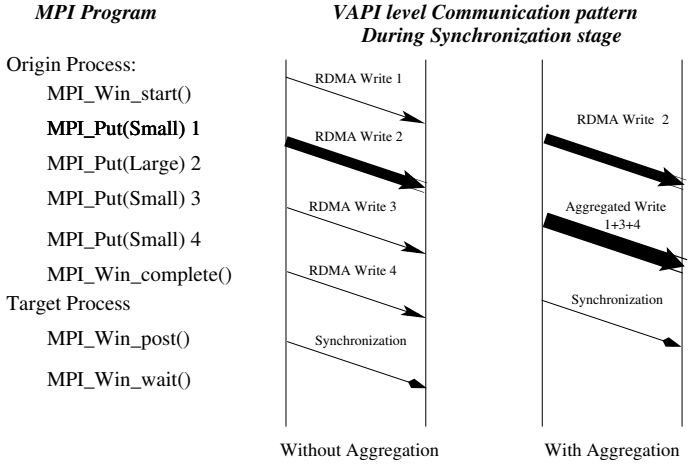

Figure 10.6: Aggregation of Multiple Small Size RMA Operations

together. The users can use MPI user defined datatypes to aggregate several one-sided and two sided operations to improve network utilization. However, our aim is to provide optimizations inside the MPI library so that we can deliver good performance even if there is no optimization at the user level. Also, as described in Section 10, no order needs to be guaranteed among the MPI_Put/MPI_Get operations between two synchronization calls. So we are not violating any MPI-2 semantics by aggregating some of these operations, as long as all the data finally reaches the target side. We can consider the following two aggregation schemes:

- Aggregation between an RMA operation and a synchronization operation
- Aggregation between multiple RMA operations

These schemes are illustrated in Figs. 10.5 and 10.6. By utilizing Point to Point Based approach, we can aggregate multiple RMA operations or an RMA operation and a synchronization operation. In contrast, Direct One Sided approach cannot provide aggregation because the target is not involved in communication and hence cannot scatter aggregated messages into target buffers. To maximize aggregation, we defer small RMA messages until we have sufficiently large number of them. Then we can trigger deferred RMA messages as an aggregated operation and send it by Point to Point Based approach. Meanwhile, large size RMA operations are still issued by Direct One Sided approach. We can also hold back one small RMA operation and combine it with the synchronization operation. In this work, we mainly focus on the aggregation between a RMA operation and a synchronization operation.

10.3 Performance Evaluation

In this section, we use several micro benchmarks to evaluate the performance of our different schemes.

Due to the lack of publicly available applications using MPI-2 one-sided calls, we came up with our own benchmarks to evaluate our scheduling schemes. We use some specific throughput and latency tests to measure the impact of our re-ordering scheme. In addition to this, we use ping-pong latency tests for MPI_Put and MPI_Get to show the benefit of the aggregation scheme.

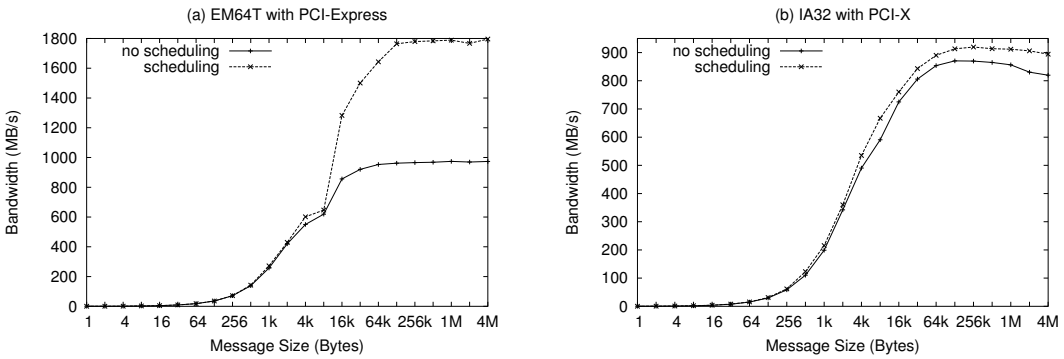

Figure 10.7: Impact of scheduling on throughput on EM64T and IA32

Experimental Testbed

We evaluated our schemes on two different testbeds. The first testbed is equipped with PCI-X interface and the second is equipped with PCI-Express interface.

Our PCI-X testbed cluster consists of 8 SuperMicro SUPER X5DL8-GG nodes with ServerWorks GC LE chipsets, Intel Xeon 3.0 GHz processors based on IA32 architecture, and PCI-X 64-bit 133 MHz bus. The PCI-Express node of our testbed

has a 3.4 GHz Intel Xeon processor based on EM64T architecture and runs in 64 bit mode with 8x PCI-Express interfaces. They are equipped with MT25208 HCAs with PCI-Express interfaces. On both platforms InfiniScale MTS2400 switch is used to connect all the nodes. The versions of InfiniBand SDK and firmware are 3.2 RC17 and 4.5.2 RC4-BUILD-001 respectively. The operating system used is RedHat Linux.

Impact of Re-ordering Scheme on different Communication Patterns

We created two communication patterns at microbenchmark level to study the impact of the re-ordering scheme we proposed in the previous section.

Communication Pattern 1

We created a throughput test which involves two processes. The first process starts a window access epoch and then issues 16 MPI_Put and 16 MPI_Get operations of the same size. The second process just starts an exposure epoch. The same sequence of operations are repeated for several iterations and we measure the maximum throughput we can achieve (in terms of MillionBytes/sec).

We compared the performance of re-ordering scheme and the original Direct One Sided implementation. On PCI-Express systems, as we can see from Fig. 10.7(a), with re-ordering scheme we are able to attain maximum throughput of 1788MB/s, which is much closer to the peak bidirectional bandwidth. We observe an improvement in throughput up to 76% compared with the original design. This trend is also there on IA32 systems where the maximum improvement of throughput is about 8%, as shown in Fig. 10.7(b). However, we do not get as much improvement as on EM64T testbed because on IA32 system, the PCI-X bus becomes the bottleneck.

Communication Pattern 2

The test consists of multiple iterations involving two processes. In each iteration, the first process calls MPI_Win_start to start a window access epoch, issues one MPI_Put and one MPI_Get, and then calls MPI_Win_complete to end the epoch. After that it starts and ends a window exposure epoch by calling MPI_Win_post and MPI_Win_wait. The second process does the same job, but in a reversed order, first it starts the exposure epoch then the access epoch. We measure the average latency for each iteration.

Our Scheduling scheme switches the order of these two operations when it is actually issuing the corresponding RDMA read or RDMA write during the access epoch. We can see that especially for large messages, we can show significant benefits by scheduling the operations internally. We can reduce the latency up to 40% on EM64T testbed and 20% on IA32 testbed, as shown in Fig. 10.8.

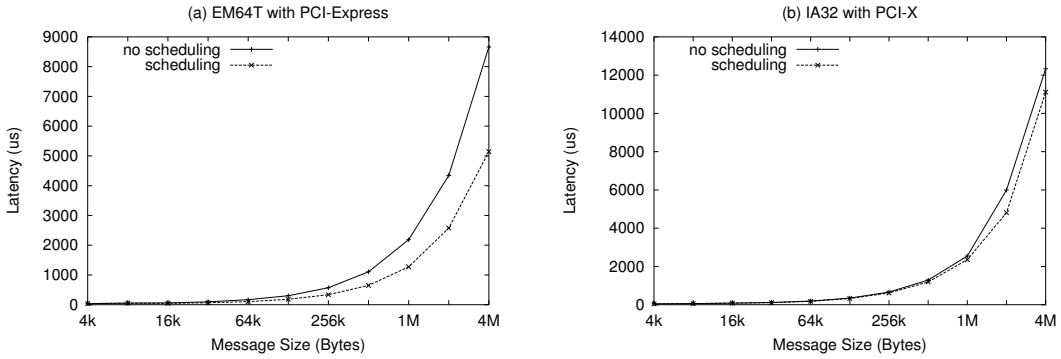

Figure 10.8: Impact of scheduling on latency on EM64T and IA32

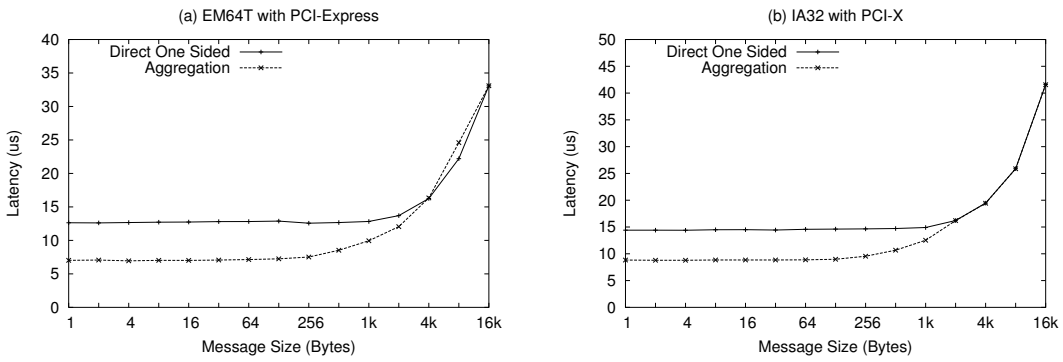

Figure 10.9: One sided MPI_Put latency on EM64T and IA32

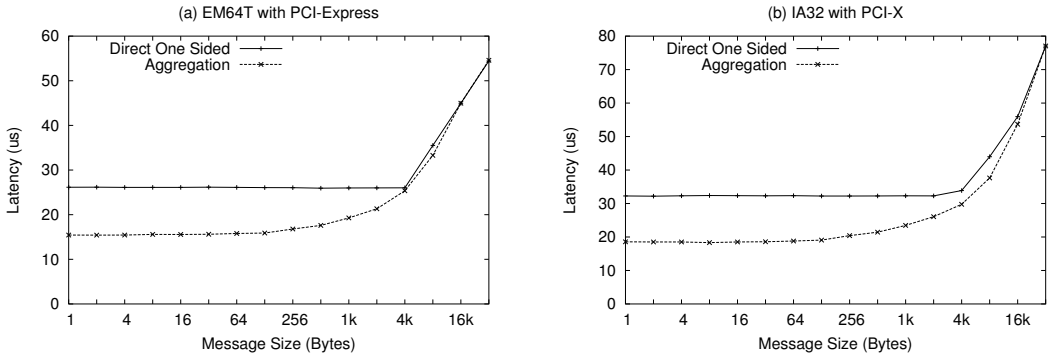

Figure 10.10: One sided MPI_Get latency on EM64T and IA32

Impact of Aggregation Scheme on Latency

In this section we measure the impact of our aggregation scheme on MPI_Put and MPI_Get latency. The test consists of multiple iterations involving two processes. In each iteration, the first process starts a window access, issues a RMA operation (MPI_Put or MPI_Get) and then ends the epoch. Then it starts and ends a window exposure epoch. The second process does the same job, but in a reversed order.

We measure the time needed for each iteration and define half of its value as the ping-pong latency for the RMA operation.

Fig. 10.9(a) compares the ping-pong latency for MPI_Put operation and Fig. 10.10(a) compares the ping-pong latency for MPI_Get operations on EM64T testbed. The aggregation scheme did noticeably better than our original Direct One Sided implementation for small size RMA operation. We see an improvement of up to 44% for MPI_Put latency and 42% for MPI_Get latency. For larger sizes, the aggregation scheme actually falls back to Direct One Sided implementation so that these two schemes delivers the same latency. We can observe the similar trends on IA32 platform, as shown in Fig. 10.9(b) and Fig. 10.10(b). The maximum improvement is around 38% and 42% for MPI_Put and MPI_Get latency respectively.

10.4 Related Work

Although we are aware of MPICH2 performing aggregation between the last one sided operation with a synchronization, to the best of our knowledge, there is no literature study on scheduling RMA operations to improve the performance of one sided implementation.

One distinguishing feature of MPI as compared to these is that MPI supports both one sided and two sided communications, which we use to our advantage in implementing our schemes. It is to be noted that ARMCI performs aggregation [48].

CHAPTER 11

SIGNIFICANCE AND IMPACT

In this thesis we have designed and developed a high performance and scalable one-sided middleware that would be beneficial to a wide range of scientific community. Specifically, we have demonstrated how we can use the features of modern interconnects to improve the performance of one-sided middleware for current and next generation High End Computing systems.

The expected contributions of the research are as follows:

- Our research demonstrates the feasibility of developing high performance and scalable one-sided communication subsystems based on the capability of modern interconnects based on the capability of modern interconnects.

- Specifically, we have demonstrated how we can leverage the advanced features like different communication semantics, remote atomic operations, completion and event mechanisms, scatter-gather support to improve performance, scalability and overlap capability for one-sided communication.

- Although we mainly concentrate on MPI one-sided communication in this work, many of our research contributions are also directly applicable to communication subsystem design in other areas such as PGAS programming models and languages.

Many of these proposed designs are being used in MVAPICH2 software which is used by more than 900 organizations worldwide and are also incorporated into a number of different vendor distributions. The MVAPICH2 software is also distributed in the OpenFabrics Enterprise Distribution (OFED). The re-ordering designs that uses prioritizing and interleaving has been integrated into MVAPICH2. The passive synchronization designs and optimizations are being integrated and will be released in the future.

CHAPTER 12

CONCLUSIONS AND FUTURE WORK

In this thesis, we have addressed the problem of providing a Scalable and High Performance Communication Middleware for one-sided communication over modern interconnects. As clusters increase in size, the performance and scalability of the communication subsystem becomes the key requirement for achieving overall scalability of the system. In this context, the efficiency of one-sided operations is especially important as they are the widely used communication operations in different programming models like MPI-2, UPC, etc. and have to be designed while harnessing the capabilities and features exposed by the underlying networks. Modern interconnects like InfiniBand provide RDMA capabilities for read/write, remote atomics, etc. These mechanisms provide good match for one-sided communication. The main issues addressed are improving computation/communication overlap, reduce remote process involvement, latency hiding mechanisms, zero-copy communication protocols, intra-node optimizations, efficient non-contiguous communication, efficient protocols for read-modify-write operations. The designs proposed in this thesis leverage the hardware primitives of modern interconnects like InfiniBand to provide good performance and scalability. The summary of the research contributions is explained in the following sections of the chapter.

12.1 Summary of Research Contributions

Improving Overlap: We investigated the designs for passive synchronization. The two-sided approaches leads to poor overlap capability. We came up with a new design using InfiniBand RDMA atomic operations to perform lock/unlock operations needed for passive synchronization. We also improved the capability of the one-sided operations to achieve faster communication progress. This work is described in Chapter 4.

Intra-node Optimizations: In this work, we designed passive synchronization mechanism for Intra-node operations using the native fast CPU based locks. We developed a hybrid design that can migrate between CPU based locks and network based locks (based on InfiniBand atomic operations). We demonstrated the benefits of the hybrid designs with various micro-benchmarks. This work is described in Chapter 5.

Synchronization optimizations: In this work, we evaluated different design alternatives for implementing fence synchronization on RDMA capable interconnects. We proposed a novel fence mechanism that uses RDMA based Immediate capability of InfiniBand to notify remote completions. This approach provides low synchronization overhead as well as good overlap capability as described in Chapter 6.

Read Modify Write Mechanisms: In this work, we studied the HPCC Random Access benchmark which predominantly uses read modify write operations. We developed one-sided versions of the random access benchmark to evaluate the read modify write capability of the MPI one-sided operations. Different optimizations like Software Aggregation and Hardware Based Accumulate were proposed to improve the GUPs rating of the HPCC benchmark. This work is described in detail in Chapter 7.

Zero-copy non-contiguous data transfer: Non-contiguous data communication poses additional challenges since it involves overhead of additional copies on the sender and receiver side. In this work, we designed zero-copy protocols using the InfiniBand hardware scatter/gather capabilities. The zero copy designs showed better performance in terms of latency and bandwidth, as well as reduced host CPU utilization. This work is described in Chapter 8.

Non-blocking Semantics: Non-blocking operations are very important to achieve latency hiding and good computation/communication overlap. In this work, we studied the issues in designing non-blocking one-sided operations in the context of ARMCI one-sided communication library. Further optimizations like capabilities for implicit and explicit aggregations were developed and the benefits of these approaches were demonstrated in Chapter 9.

Re-ordering one-sided operations: The MPI one-sided semantics allow re-ordering of the one-sided operations within an access epoch. Maintaining the order of operations does not always lead to the best or optimum usage of the underlying network capability. In this work we exploited this flexibility to explore different techniques like interleaving, prioritizing and aggregation to reorder these RMA operations based on the communication pattern to improve the latency, bandwidth and throughput. This work is described in Chapter 10.

12.2 Future work

- **Intra-Node Optimizations for Reducing Copy Costs:** With the advent of multicore processor technology, a large number of processing cores can reside within one node, increasing the number of MPI processes inside a node, thus increasing the volume of communication within the node. Therefore, designing an one-sided library with optimized intra-node communication support is crucial to overall performance. The current shared memory approach for inter-node communication needs two copies. One optimization is to use the kernel (using approaches like LIMIC [38]) to copy directly into the target window to reduce the copy overhead. Another approach could be to use IOAT offload engines to offload the copy operations.

- **Application level Evaluation:** As part of future work, applications need to be written with one-sided semantics. Specifically we would like to target some of the communication patterns of petascale applications like AWM-Olsen [68] (earthquake simulation) and MPCUGLES [58] (Computational Fluid Dynamics Code). These applications need support from the middleware in terms of rich interface to express parallelism, good computation/communication overlap and dynamic load balancing. Currently these applications are written using two-sided communication. These applications have the potential to exploit one-sided communication to attain petascale performance and scalability.

- **Propose Extensions to MPI One-sided Semantics:** In order to handle some of the requirements from these petascale applications, additional support and enhancements might be needed from the communication subsystem or

middleware. We have identified some limitations of existing MPI-2 one-sided semantics. Extensions to one-sided semantics can be proposed to address some of these issues. Some of these extensions could be aimed at providing improved and more flexible/(less restrictive) synchronization semantics for both active and passive synchronization. In case of Passive synchronization, applications could benefit from finer grain locking semantics. In case of Active synchronization, fence synchronization that are targeted towards specific communication patterns would be beneficial as that can result in lower overheads. Furthermore, non-blocking synchronization primitives can allow applications to exploit computation and communication overlap. Additional interfaces can also be provided that can aid dynamic load balancing and fault tolerance which are critical for applications to scale.

BIBLIOGRAPHY

[1] A Generalized Portable SHMEM library.

[2] Berkeley Unified Parallel C (UPC) Project. http://upc.lbl.gov/.

[3] Direct Numerical Simulation (DNS). http://www.cfd-online.com/Wiki/Direct_numerical_simulation_(DNS).

[4] Global Arrays. http://www.emsl.pnl.gov/docs/global/.

[5] GROMACS. http://www.gromacs.org/.

[6] Intel 80 core Teraflops chip. http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

[7] Mellanox InifniBand Technologies. http://www.mellanox.com.

[8] PETSc. http://www-unix.mcs.anl.gov/petsc/.

[9] Argonne National Laboratory. MPICH2 Release 0.96p2. `http://www-unix.mcs.anl.gov/mpi/mpich2/`, Jan 2004.

[10] Mike Ashworth. A Report on Further Progress in the Development of Codes for the CS2. In *Deliverable D.4.1.b F. Carbonnell (Eds), GPMIMD2 ESPRIT Project, EU DGIII, Brussels*, 1996.

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

[12] B. W. Barrett, G. M. Shipman, and A. Lumsdaine. Analysis of Implementation Options for MPI-2 One-Sided. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[13] MPI-2 One-Sided based HPCC Random Access benchmarks. http://nowlab.cse.ohio-state.edu/projects/hpcc-one-sided/.

[14] Christian Bell and Dan Bonachea. A new dma registration strategy for pinning-based high performance networks. *ipdps*, 00:198a, 2003.

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

[16] S. Booth and F. E. Mourao. Single Sided MPI Implementations for SUN MPI. In *Supercomputing*, 2000.

[17] D. Buntinas, D. K. Panda, and W. Gropp. NIC-Based Atomic Remote Memory Operations in Myrinet/GM. Workshop on Novel Uses of System Area Networks (SAN-1), February 2002.

[18] Surendra Byna, Xian-He Sun, William Gropp, and Rajeev Thakur. Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[19] Cristian Coarfa, Yuri Dotsenko, and John Mellor-Crummey. Experiences with sweep3d implementations in co-array fortran. *J. Supercomput.*, 36(2):101–121, 2006.

[20] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The design, implementation, and evaluation of mpiblast. In *Proceedings of ClusterWorld 2003*, 2003.

[21] A. Devulapalli and P. Wyckoff. Distributed queue based locking using advanced network features. In *ICPP*, 2005.

[22] Jack Dongarra and Piotr Luszczek. overview of the hpc challenge benchmark suite. SPEC Benchmark Workshop, 2006.

[23] M. Frigo and S.G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 2005.

[24] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modelling Astro physical Thermonuclear Flashes. *Astrophysical Journal Suppliment*, 131:273, 2000.

[25] G. Bryan. Fluid in the universe: Adaptive Mesh Refinement in cosmology. In *Computing in Science and Engineering*, volume 1, pages 46–53, March/April 1999.

[26] Rahul Garg and Yogish Sabharwal. Optimizing the HPCC randomaccess benchmark on blue Gene/L Supercomputer. ACM SIGMETRICS Performance Evaluation Review, June 2006.

[27] R. A. Van De Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[28] 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.

[29] W. Gropp and E. Lusk. A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.

[30] 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.

[31] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.

[32] William Gropp, Ewing Lusk, and Deborah Swider. Improving the Performance of MPI Derived Datatypes. In *MPIDC*, 1999.

[33] William D. Gropp and Rajeev Thakur. An evaluation of implementation options for mpi one-sided communication. In *PVM/MPI*, pages 415–424, 2005.

[34] P. Husbands and J. C. Hoe. MPI-StarT: Delivering Network Performance to Numerical Applications. In *Proceedings of the Supercomputing*, 1998.

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

[36] J. L. Träff, H. Ritzdorf and R. Hempel. The Implementation of MPI–2 One-sided Communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.

[37] W. Jiang, J.Liu, H. W. Jin, D. K. Panda, D. Buntinas, R.Thakur, and W.Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. EuroPVM/MPI, September 2004.

[38] Hyun-Wook Jin and Dhabaleswar K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 184–191, Washington, DC, USA, 2005. IEEE Computer Society.

[39] J.Nieplocha, V.Tipparaju, and E.Apra. An evaluation of two implementation strategies for optimizing one-sided atomic reduction. International Parallel and Distributed Processing Symposium, 2005.

[40] J. Liu, W. Jiang, Hyun-Wook Jin, D. K. Panda, W. Gropp, and Rajeev Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. International Symposium on Cluster Computing and the Grid (CCGrid 04), April 2004.

[41] Qingda Lu, Jiesheng Wu, Dhabaleswar K. Panda, and P. Sadayappan. Employing MPI Derived Datatypes to the NAS Benchmarks: A Case Study . Technical Report OSU-CISRC-02/04-TR10, Dept. of Computer and Information Science, The Ohio State University, Feb. 2004.

[42] A. Mamidala, S. Narravula, A. Vishnu, G. Santhanaraman, and D. K. Panda. On using Connection-Oriented and Connection-Less transport for Performance and Scalability of Collective and One-sided operations: Trade-offs and Impact. In *PPoPP*, 2007.

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

[44] F. E. Mourao and J. G. Silva. Implementing MPI's One-Sided Communications for WMPI. In *EuroPVM/MPI*, September 1999.

[45] S. Narravula, A. Mamidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High performance distributed lock management services using network-based remote atomic operations. CCGrid, 2007.

[46] Network-Based Computing Laboratory. MPI over InfiniBand Project. http://mvapich.cse.ohio-state.edu/.

[47] 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.

[48] Jarek Nieplocha, Vinod Tipparaju, Manoj Krishnan, Gopalakrishnan Santhanaraman, and Dhabaleswar K. Panda. Optimizing Mechanisms for Latency Tolerance in Remote Memory Access Communication on Clusters . In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[49] Jarek Nieplocha, Vinod Tipparaju, Amina Saify, and Dhabaleswar Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. *ipdps*, 2:0164, 2002.

[50] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, pages 340–ff., Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[51] R.J.Thacker, G.Pringle, H.M.P Couchman, and S.Booth. Hydra-mpi: An adaptive particle-particle, particle-mesh code for conducting cosmological simulations on mpp architectures. High Performance Computing Systems and Applications, 2003.

[52] Robert Ross, Neill Miller, and William Gropp. Implementing Fast and Reusable Datatype Processing. In *EuroPVM/MPI*, Oct. 2003.

[53] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and Experience with LAPI: A New High Performance Communication Library for the IBM RS/6000 SP. In *Proceedings of International Parallel Processing Symposium*, 1998.

[54] J. P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. *SIGARCH Comput. Archit. News*, 20(1):5–44, 1992.

[55] 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.

[56] HPCC Benchmark Suite. http://icl.cs.utk.edu/hpcc.

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

[58] Mahidhar Tatineni and Mahidhar Tatineni. SDSC HPC Resources. https://asc.llnl.gov/alliances/2005_sdsc.pdf.

[59] T.El-Ghazawi, F.Cantonnet, Y.Yao, and J.Vetter. Evaluation of UPC on the Cray X1. Cray User Group meeting, 2006.

[60] R. Thakur, W. Gropp, and B. Toonen. Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication. In *EuroPVM/MPI*, September 2004.

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

[62] Vinod Tipparaju, Manoj Krishnan, Jarek Nieplocha, Gopalakrishnan Santhanaraman, and Dhabaleswar K. Panda. Exploiting nonblocking remote memory access communication in scientific benchmarks . In *Proceedings of the International Conference on High performance Computing (HiPC 03)*, 2003.

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

[64] Randolph Wang, Arvind Krishnamurthy, Richard P. Martin, Thomas E. Anderson, and David E. Culler. Modeling communication pipeline latency. In *Measurement and Modeling of Computer Systems*, pages 22–32, 1998.

[65] J. B. White and S. W. Bowa. Where's the overlap? overlapping communication and computation in several popular mpi implementations. *Proceedings of the Third MPI Developers and Users conference*, 1999.

[66] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Supporting Efficient Noncontiguous Access in PVFS over InfiniB and. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2003.

[67] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance Implementation of MPI Datatype Communication over InfiniBand. In *International Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

[68] Y. Cui, R. Moore, K. Olsen, A. Chorasia, P. Maechling, B. Minister, S. Day, Y. Hui, J. Zhu, A. Majumdar and T. Jordan. Enabling very large earthquake simulations on Parallel Machines. In *Lecture Notes in Computer Science*, 2007.