ENHANCING MPI POINT-TO-POINT AND COLLECTIVES FOR CLUSTERS WITH ONLOADED/OFFLOADED INFINIBAND ADAPTERS

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Rahul Kumar, B.Tech.

* * * * *

The Ohio State University

2008

Master's Examination Committee:

Prof. Dhabaleswar K. Panda, Adviser

Prof. P. Sadayappan

Approved by

Adviser

Graduate Program in
Computer Science and
Engineering

ABSTRACT

Many applications from various fields such as life sciences, weather forecasting, financial services require massive amounts of computational power. Supercomputers built using commodity components, called clusters, are a very cost effective way of providing such huge computational power. Recently, the supercomputing arena has witnessed phenomenal growth of commodity clusters built using InfiniBand interconnects and multi-core systems. InfiniBand is a high performance interconnect providing low latency and high bandwidth. Message Passing Interface (MPI) is a popular model to write applications for such machines. Therefore, it is important to optimize MPI for these emerging clusters.

InfiniBand architecture allows for varying implementations of the network protocol stack. For example, the protocol can be totally on-loaded to the host processing core or it can be off-loaded onto the NIC processor or can use a combination of the two. Understanding the characteristics of these different implementations is critical in optimizing MPI. In this thesis, we systematically study some of these architectures which are commercially available. Based on their characteristics, we propose communication algorithms for one of the most extensively used collective operation, MPI_Alltoall. We also redesign the point-to-point rendezvous protocol for offload network interfaces to allow for overlap of communication and computation. The designs developed as part of this thesis are available in MVAPICH, which is a popular

open-source implementation of MPI over InfiniBand and is used by several hundred top computing sites all around the world.

This is dedicated to my parents

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for guiding me throughout the duration of my M.S. study. I am thankful for all the efforts he took for my thesis. I would like to thank him for all his advices during my stay here. I am thankful to Prof. P. Sadayappan for agreeing to serve on my Masters examination committee.

I am grateful for the financial support by Qlogic and Mellanox. I am especially thankful to Dr. Amith Mamidala, who was not only a great mentor, but a close friend. I am grateful to have had Dr. Sayantan Sur as a mentor during my first year of graduate study. I am also thankful to Matthew Koop for his guidance and discussions.

I would like to thank all my senior Nowlab members Gopal Santhanaraman, Dr. Karthik Vaidyanathan, Sundeep Narravula, Ranjit Noronha, Wei Huang and Lei Chai for their patience and guidance. I would also like to thank all my colleagues Ping Lai, Debraj De, Xiangyong Ouyang, Karthik Gopalakrishnan, Jaidev Sridhar and Tejus Gangadharappa. I would also like to thank Jonathan for helping me in solving my equipment problems.

I would also like to thank all the people who made my stay at Ohio State memorable especially Sagar and Jatin.

I would like to thank my family members, Renuka(my mom), Dilip (my dad), Rohit (my brother), Rohini and Ragini (my sisters) for their love and support. Finally,

I would also like to thank my uncles Ganga Sah and Shambhu Sah without them I would not be what I am today.

VITA

September 12, 1981	Born - Madhubani, India
2004	B.Tech. Electrical Engineering, Indian Institute of Technology (IIT) Roorkee, India
2004-2006	Applications Engineer, Oracle Corporation.
2006-present	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

Research Publications

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

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

R. Kumar, A. Mamidala and D. K. Panda "Scaling Alltoall Collective on Multi-core Systems". *Workshop on Communication Architecture for Clusters (CAC); in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS)*, Miami, Florida, April, 2008.

A. Mamidala, R. Kumar, Debraj De and D. K. Panda "MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics". *Int.l Symposium on the Cluster Computing and the Grid (CCGrid)*, Lyon, France, May, 2008.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Studies in High Performance Computing Prof. D. K. Panda

TABLE OF CONTENTS

Page

Abstract ii

Dedication iv

Acknowledgments v

Vita vii

List of Figures xi

Chapters:

1. Introduction 1

    1.1 Overview of InfiniBand Architecture 2
        1.1.1 Offload InfiniBand Interfaces 4
        1.1.2 Onload InfiniBand Interfaces 6
    1.2 Overview of MPI 7
        1.2.1 Point-to-point 7
        1.2.2 Collective 8
    1.3 Problem Statement 8
    1.4 Our Approach 10

2. Asynchronous Progress for Offload Interfaces 13

    2.1 Background 14
        2.1.1 Point-to-Point Communication Protocols 14
        2.1.2 Progress 16
    2.2 Related Work 17
    2.3 Design of Asynchronous Progress 19

2.4 Evaluation . . . 21
2.4.1 Comparison with existing design . . . 21
2.4.2 Overlap Microbenchmark Performance . . . 23
2.4.3 Application Performance . . . 25
2.5 Summary . . . 28

3. Alltoall Collective for Onload and Offload Interfaces . . . 29

3.1 Background . . . 30
3.1.1 Alltoall Algorithms . . . 30
3.1.2 CPMD . . . 31
3.2 Motivation . . . 32
3.2.1 Performance of bi-directional bandwidth . . . 33
3.3 Related Work . . . 35
3.4 Proposed Design . . . 36
3.5 Evaluation . . . 39
3.5.1 Alltoall Performance . . . 40
3.5.2 Application Performance . . . 43
3.6 Summary . . . 45

4. Contributions and Future Work . . . 46

4.1 Summary of Research Contributions and Future Work . . . 46
4.1.1 Providing Asynchronous Progress in Rendezvous Protocol . 46
4.1.2 Improving Performance of Alltoall Collective . . . 47

Appendices:

A. Scheduling Policies . . . 49

Bibliography . . . 52

LIST OF FIGURES

Figure **Page**

2.1 Rendezvous protocols 15

2.2 Overlap capability of rendezvous protocols 17

2.3 Asynchronous Rendezvous Protocol Implementations 19

2.4 Bandwidth for large messages 21

2.5 NAS-SP Normalized Execution Time 21

2.6 Application availability at Receiver 23

2.7 Application availability at Sender 23

2.8 Effect of scheduling algorithm 24

2.9 Matrix Multiplication: MPI + OpenMP configuration 26

2.10 Matrix Multiplication: MPI x1 configuration 26

2.11 Matrix Multiplication: MPI x4 configuration 27

3.1 Performance of alltoall on multi-core systems 32

3.2 InfiniPath SDR: Multi-pair Bidirectional Bandwidth. 33

3.3 ConnectX DDR: Multi-pair Bidirectional Bandwidth. 34

3.4 InfiniHost III DDR: Multi-pair Bidirectional Bandwidth. 35

3.5 Communication steps of the proposed design 37

3.6 InfiniPath: Alltoall time on 64X8 system 40

3.7 InfiniPath: Alltoall time of 512Byte message 42

3.8 Infinihost III: Performance of different schemes on 64X8 system . . . 42

3.9 ConnectX: Performance of different schemes on 4X8 system 43

3.10 CPMD Application Benchmark Performance on InfiniPath: Different Input Files 44

3.11 CPMD Application Benchmark Performance on InfiniPath: Varying System Sizes 45

CHAPTER 1

INTRODUCTION

Large computing power is required to solve many applications in different fields such as life sciences, financial services, engineering and biotechnology. Due to its cost effectiveness, building a supercomputer from commercially available hardware components has never been as popular as today. The current fastest supercomputer, codenamed Roadrunner[1], is a paramount example to it. This is the first time that a cluster is the fastest supercomputer in the world and in fact, it is twice as fast as the next fastest supercomputer. A cluster is a machine built out of commodity components taken off-the-shelf.

Cluster computing is also observing a dramatic increase in size due to the advent of multi-core architecture. Multi-core architecture is a growing industry trend as single core processors are reaching the physical limits in terms of complexity, speed and thermal limit. Many new deployments in high-end computing systems are multi-core based, employing InfiniBand[13] as the cluster interconnect.

InfiniBand is a high-performance cluster interconnect based on open standards. It offers several features to achieve very low communication latency and high bandwidth required by many applications. It has gained widespread acceptance in high performance computing over the last few years. This is evident by the increase in number

of top 500[5] supercomputers using InfiniBand as the interconnect. Also, over the past few years there has been a surge in network interface based processing (offload) networks. But with change in processor architectures to multi-core processor, host based NICs have re-emerged to take advantage of faster processing power of on-board cores. This has renewed the discussion on offload and onload[29] network processing.

Message Passing Interface (MPI)[19] is the most commonly used method for programming high performance computing (HPC) systems. MVAPICH[21] is a popular MPI implementation over InfiniBand. It is used by many organizations and powers several of the top 500 supercomputers.

In this chapter we provide an overview of InfiniBand interconnect and message passing interface (MPI).

1.1 Overview of InfiniBand Architecture

Many HPC applications are latency and/or bandwidth sensitive. Traditional networks like Ethernet were not able to match the tremendous increase in computing speed of modern processors. This caused communication as the bottleneck in scaling to larger systems. There was a need for high-performance network. InfiniBand (IB) was originally proposed as a general I/O technology, allowing for a single fabric to replace all existing fabrics. However, currently InfiniBand is mainly used as an Inter Process Communication (IPC) and Storage Area Network (SAN) interconnect technology.

The main reasons for high latency and low bandwidth of traditional networks were unnecessary intermediate buffer copies and context switches between user and kernel mode during each communication. In order to avoid multiple context switches

and buffer copies, InfiniBand provides Operating System (OS) bypass facility. In InfiniBand, unlike traditional stack based protocols, the OS is not involved in segmentation or processing of other protocol specific messages. To enable OS bypass, InfiniBand defines the concept of Queue Pair (QP). The Queue Pair model provides user level processes direct access to the IB Host Channel Adapter (HCA). Each queue pair consists of both send and receive work queue, and is additionally associated with a Completion Queue (CQ). Work Queue Entries (WQEs) are posted from the user level for processing by the HCA. Upon completion of a WQE, the HCA posts an entry to the completion queue, allowing the user level process to poll and/or wait on the completion queue for events related to the queue pair.

In addition to send/receive communication semantics, InfiniBand also provides Remote Direct Memory Access (RDMA) capability. RDMA enables data transfer from the address space of an application process to another process across the network fabric without requiring involvement of the host CPU. This removes unnecessary storage of data in intermediate buffers.

Two-sided send/receive operations are initiated by enqueueing a send WQE on a QP send queue. The WQE specifies only the senders local buffer. The remote process must pre-post a receive WQE on the corresponding receive queue which specifies a local buffer address to be used as the destination of the receive. Send completion indicates the send WQE is completed locally and results in a sender side CQ entry. When the transfer actually completes a CQ entry will be posted to the receivers CQ. One-sided RDMA operations are likewise initiated by enqueueing a RDMA WQE on the Send Queue. However, this WQE specifies both the source and target virtual addresses along with a protection key for the remote buffer. Both the protection key

and remote buffer address must be obtained by the initiator of the RDMA read/write prior to submitting the WQE. Completion of the RDMA operation is local and results in a CQ entry at the initiator. The operation is one sided in the sense that the remote application does not receive notification of its completion. However, this places some additional constraints on the source/destination buffers involved in the transfer. As data is moved directly between the host channel adapter (HCA) and user level source/destination buffers, these buffers must be registered with the HCA in advance of their use. Registration is a relatively expensive operation which locks the memory pages associated with the request, thereby preserving the virtual to physical mappings.

Additionally, when supporting send/receive semantics, preposted receive buffers are consumed in order as data arrives on the host channel adapter (HCA). Since no attempt is made to match available buffers to the incoming message size, the maximum size of a message is constrained to the minimum size of the posted receive buffers. The network processing, such as segmentation, assembly, reliability, transfer of data from main memory to NIC, can either be offloaded to the NIC or handled by the host processor itself.

1.1.1 Offload InfiniBand Interfaces

Offload interfaces relieve the host processor of network processing. This enables concurrent processing of computation and network. These devices feature an HCA core that is capable of performing various network tasks such as:

- Remote Direct Memory Access (RDMA): Direct access to remote memory eliminates the need to copy data, which reduces CPU overhead as well as latency and saves host memory bandwidth.

- Operating System Bypass: This feature enables applications to eliminate kernel calls to the operating system, which greatly reduces CPU overhead, context switching overhead and latencies.

- Transport Offload Hardware: Enables reduced CPU overhead.

There are many such adapters commercially available. The most popular are InfiniHost III[2] and ConnectX[18] architectures provided by Mellanox Technologies[4]. InfiniHost III is the third generation of InfiniBand Host Channel Adapter (HCA) from Mellanox. It features a full hardware implementation of the InfiniBand architecture with Hardware Transport Engine that drastically reduces the host CPU overhead on communication. ConnectX is the fourth generation InfiniBand HCA from Mellanox. The ConnectX architecture is designed to improve the processing rate of incoming packets. Compared to the previous InfiniHost III architecture, it has more advanced packet processing capabilities. In order to effectively use these capabilities, ConnectX has advanced scheduling engines which can assign processing duties (like protocol processing, data integrity checks, etc.) to idle processing elements. The scheduling of packet processing is done directly in hardware, without firmware involvement in the critical path. These enhancements to the ConnectX architecture are expected to improve its performance on multi-core nodes when multiple processes are communicating at the same time, generating many simultaneous network messages. For very small message sizes (less than around 512Bytes), Programmed I/O (PIO) is used to send

data to the network interface. This is different from the InfiniHost III architecture which uses DMA for all message sizes.

1.1.2 Onload InfiniBand Interfaces

Onload interfaces do not contain an embedded processor and all control and protocol stack operations are performed on the host processor. The advantage of this architecture is the host processor is much more powerful computationally and hence makes it possible to handle protocol computations much faster than an embedded processor. The disadvantage is that while the host is processing communication protocols it is not available to perform application processing. There can be various levels of onloading depending on the kind of network tasks performed by the host processor instead of NIC. Higher increase in core counts relative to the improvements in adapter clock frequency may provide justification for a higher ratio of host-based to adapter-based protocol handling.

Currently there is only one InfiniBand adapter available, called InfiniPath[27], which uses this principle. InfiniPath does not use DMA engines to transfer data from the memory to NIC, but rather accesses interface memory via programmed I/O. This means that the interface does not need to explicitly validate and map memory for transmits. Therefore it does not require memory used for data transmits to be registered or pinned with the network interface. Unlike traditional InfiniBand adapters, InfiniPath is stateless. Traditional IB adapters not only require an explicit connection, but they also require that some application memory be committed to a connection in order to transfer data. In order to support a model like MPI, this can lead to an extremely large amount of memory. This makes InfiniPath more scalable.

1.2 Overview of MPI

MPI[19] has established itself as the de-facto standard of parallel computing. Nearly all scientific computation applications are written using MPI, and many higher-level communication libraries require MPI. MPI is very portable and has been ported to nearly all parallel computer architectures. There is a wide variety of quality MPI implementations available as open-source: MPICH [15], MPICH2 [27], MVAPICH, MVAPICH2 [34] and OpenMPI [13]. MPI provides two major modes of communication, point-to-point and collective. In point-to-point communication, individual pairs of processes are involved in sending and receiving messages. In collective communication operations, groups of processes are involved in the data exchange. In this section, we describe the major communication modes offered by MPI in detail, their semantics and issues in designing high-performance MPI.

1.2.1 Point-to-point

In an MPI program, two processes can exchange messages using point-to-point communication primitives. The process wishing to send a message, can send it using a function MPI_Send. The receiving process may retrieve this message with a matching MPI_Recv. Messages are matched by a three-tuple source, tag and context. The source indicates the process where the message originated. The tag is a user supplied integer value and can be used to separate different messages. The context is the group of the processes the sending process belongs to. MPI_Send and MPI_Recv are the most commonly used MPI functions. However, there are variations of these calls. MPI_Send and MPI_Recv are often called as the blocking mode calls, i.e. the sending and receiving processes block on these calls until the corresponding operations

complete, or the message buffers can be reclaimed by the application. MPI_Isend and MPI_Irecv are the asynchronous versions of the send and receive calls. Using these, the application can initiate send and receive operations while continuing to perform its own computation. The MPI library will attempt to make progress in the meanwhile and can complete these operations. In order to finish the asynchronous operations, the application then needs to call MPI_Wait.

1.2.2 Collective

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

1.3 Problem Statement

Different applications exhibit different characteristics. Just as one size does not fit all, a single MPI design for different architectures of InfiniBand interconnect does not achieve the maximum performance or scalability. Onload and offload interconnect architectures offer many benefits which can help certain applications. It is necessary for the MPI designer to provide these benefits to the applications.

Offload interfaces are capable of handling communication without the intervention of CPU. Applications can take benefit of this by overlapping communication with computation. However, this ability needs to be provided by the MPI implementation. MPI does provide non-blocking semantics so that the application can benefit from computation and communication overlap. However, it requires some support from hardware in order to achieve this overlap. Certain offload adapters, such as Quadrics[28], have the hardware capability to provide this without help from MPI. However, most of the InfiniBand adapters do not have this capability due to hardware complexity. Typical MPI implementations use eager protocol for short messages and provide for good amount of overlap. For large messages, typically rendezvous protocol is used. The rendezvous protocol involves a handshake to negotiate buffer availability and then the message transfer takes place. Although this protocol provides for good buffer utilization, it hinders the ability to overlap communication with computation. MPI must provide for asynchronous progress of the rendezvous protocol, in order to enable overlap of large message communication. Thus it is important to address this issue in MPI implementation for offload adapters. In particular, we aim to provide answers to the following questions:

- What are the drawbacks of current asynchronous progress designs?
- Can a better asynchronous progress be designed?
- Can asynchronous progress provide overlap without performance degradation?

In addition to point-to-point communication, collectives are extensively used in applications. Among them complete data exchange collective, MPI_Alltoall, is one of the most intensive communication patterns used in various applications including

molecular dynamics applications like CPMD [3], NAMD [26], LU-factorization, FFT and matrix transpose. MPI_Alltoall is known to suffer from performance scaling problems. With the increase in the number of processing elements, owing to multi-core systems, it is highly desirable to optimize this data intensive communication primitive. Several algorithms have been proposed to optimize this collective in the past. With the introduction of new architectural concepts such as multi-core processors, these algorithms need renewed attention. In multi-core systems, the processes within a node have a very low latency communication between them compared to inter-node latency. This gives scope for improved algorithms. The problem also requires study with respect to different network architectures such as offloading and onloading of network protocol.

In this work, we aim to provide answers to the following questions:

- How do the current multi-core aware algorithms for other collectives suit for alltoall?
- What are the characteristics of offload and onload InfiniBand NICs?
- Are collectives algorithms affected by offload/onload architecture?
- Are there better alltoall collective algorithms for these NICs?

1.4 Our Approach

There are several designs that have been proposed previously to provide asynchronous progress. These designs typically use an additional thread to handle incoming rendezvous requests. For example, in [34], a RDMA read based threaded design is proposed to provide asynchronous progress. Though the basic approach has been

proven to achieve good computation and communication overlap, there are several overheads associated with the implementation of the design. First, the existing design uses locking to protect the shared data structures in the critical communication path. Second, it uses multiple interrupts to make progress. Third, there is no mechanism to selectively ignore the events generated. In this work, we propose an enhanced asynchronous rendezvous protocol which overcomes these limitations. Specifically, our design does not require locks in the communication path. In our approach, the main application thread makes progress on the rendezvous transfer with the help of an additional thread. The communication between the two threads occurs via system signals.

In the second work, we study the characteristics of offload and onload NICs and demonstrate how algorithms for MPI_Alltoall is affected by these characteristics. Specifically, we propose different algorithms for multi-core systems connected with varying implementations of InfiniBand network interfaces.

The rest of the thesis is organized in the following way. In Chapter 2, we redesign the rendezvous protocol to provide for overlap of communication with computation. We first study and evaluate the current designs for asynchronous progress of rendezvous protocol. We propose an improved design and evaluate our design using publicly available benchmarks. Finally, we show the performance improvement of matrix multiplication kernel with the proposed design by overlapping communication with computation.

In Chapter 3, we take on the challenge of developing better algorithms for alltoall collective. First we study the current alltoall algorithms. Then we evaluate the

network characteristics of different modern NICs. Based on this we propose different algorithms for different NICs. Final conclusions and areas for future work are presented in Chapter 4.

CHAPTER 2

ASYNCHRONOUS PROGRESS FOR OFFLOAD INTERFACES

Offload interfaces can transfer data from memory without the involvement of the host processor. Applications can benefit from this by concurrently performing computation after initiation of communication. MPI does provide non-blocking semantics of point-to-point communication. Applications can use MPI_Isend, MPI_Irecv to initiate the communication operations and return to computing. When it needs the message, the application can call MPI_Wait. Most high-performance MPI implementations are based on polling progress engines, i.e. the sender and receiver processes must periodically call MPI functions to ensure communication progress. However, due to certain MPI internal protocols (such as rendezvous protocol), overlap of computation and communication may be hampered. If progress calls are not triggered for a long time, messages may be severely delayed. In this Chapter, we take on the challenge of redesigning the rendezvous protocol in order to improve the computation and communication overlap. We design a lock-free asynchronous progress engine for RDMA based rendezvous protocol.

The rest of the chapter is organized as follows. In Section 2.1, we provide necessary background information for this work. In Section 2.2, we describe current approaches

to provide asynchronous progress in MPI and their limitations. In Section 2.3, we discuss our proposed design. In Section 2.4, we evaluate our design and provide experimental results. Finally, in Section 2.5, we summarize the results and impact of this work.

2.1 Background

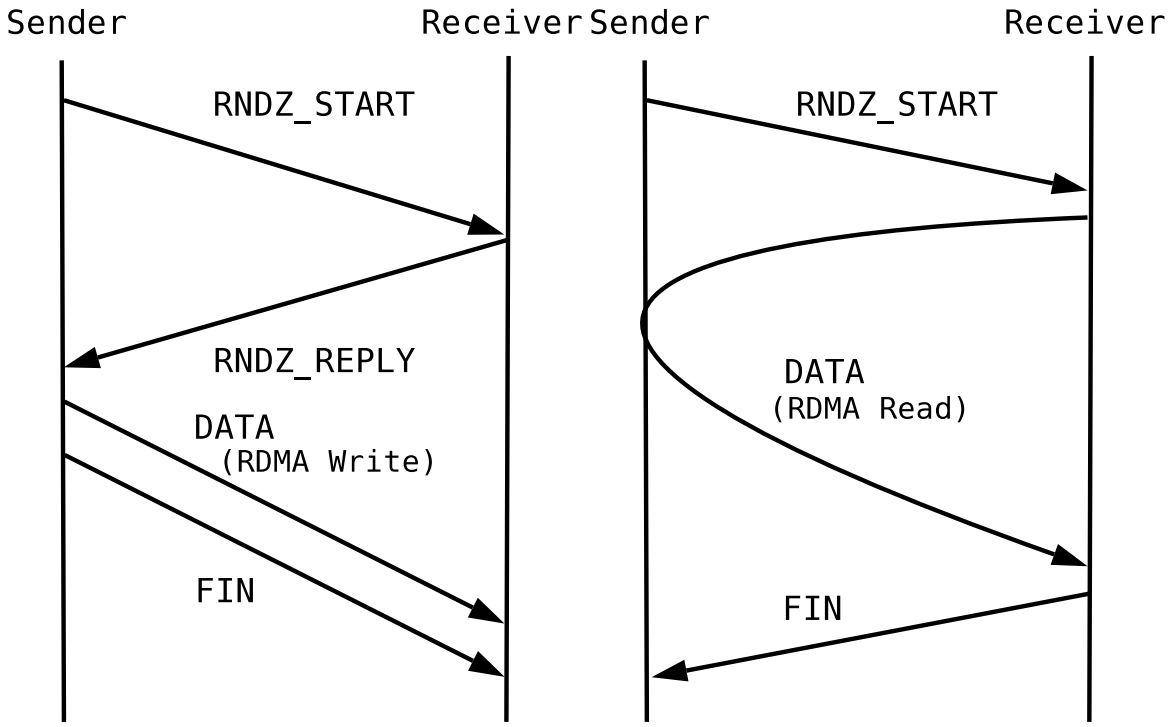
In this Section, we provide the necessary background on MPI point-to-point protocols and the need for asynchronous progress.

2.1.1 Point-to-Point Communication Protocols

MPI communication is often implemented using two general protocols:

Eager protocol: In this protocol, the sender process sends the message eagerly to the receiver. The receiver needs to provide buffers in advance for the incoming messages. This protocol has low startup overhead and is used for small messages.

Rendezvous protocol: The rendezvous protocol involves a handshake during which the buffer availability is negotiated. The message transfer occurs after the handshake. This protocol is used for transferring large messages. In the rendezvous protocol, the actual data can be transferred using RDMA operations or using send-recv operations. In the send-recv method, the messages are divided into small chunks and transmitted using eager protocol. At the receiver, the chunks are assembled into the destination buffer. This protocol, however, incurs additional message copies. In the other method, RDMA write or RDMA read can be used to transfer the data. Both RDMA based approaches can achieve zero copy message transfer. The RDMA write based and RDMA read based rendezvous protocol have been shown in Figures 2.1(a) and 2.1(b), respectively.



(a) RDMA write based

(b) RDMA read based

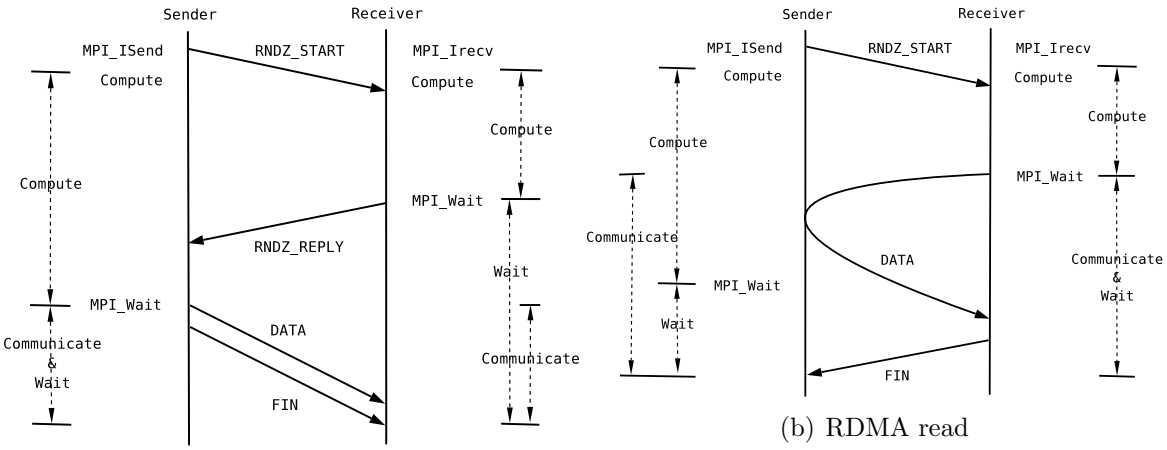
Figure 2.1: Rendezvous protocols

2.1.2 Progress

RDMA based rendezvous protocols are more popular for modern networks, which support RDMA operations. OS bypass communication architecture, such as in InfiniBand, allow the network interface card to perform data transfer asynchronously with the host processor. This also allows for overlap of communication with computation. MPI provides non-blocking communication primitives to take advantage of this capability of networks. However, for large messages, which use rendezvous protocol, the ability to leverage overlap is limited due to lack of asynchronous progress of the rendezvous protocol.

Suppose the receiving process posts a non-blocking receive before the arrival of the RNDZ_START message. After the post, the receiver begins to perform computation. On arrival of the RNDZ_START message, the receiver is unaware of it's arrival. The progress on the communication takes place after the receiver finishes the computation. This is also shown in Figures 2.2(a) and 2.2(b) for RDMA write and RDMA read based protocols, respectively. Therefore, most of the communication takes place within the MPI_Wait call when the host processor is unable to perform any useful work. In the RDMA write based protocol, both the sender and receiver are unable to provide for overlap. In RDMA read based protocol, the receiver is unable to achieve any overlap. However, the sender is able to achieve almost total overlap as it only takes part in sending the initial request message.

Asynchronous progress is useful for both onload as well as offload interfaces to help the sender and receiver process communication independently. In offload interfaces, it also helps in achieving overlap of communication with computation.



(a) RDMA write

(b) RDMA read

Figure 2.2: Overlap capability of rendezvous protocols

2.2 Related Work

Several studies have been done to show the importance of overlap capability in MPI library. Brightwell et al. [6] show the impact of these features on applications. Eicken et al. [8] propose for hardware support for active messages to provide communication and computation overlap. Schemes to achieve overlap in one-sided communication have been proposed in [22]. Sur et al. [34] propose thread based rendezvous protocol which employs locks for protection. Our design is an improvement over this design.

In this Section, we explain the design proposed in [34]. The design used InfiniBand's RDMA read capability together with IBA's event notification mechanism. Figure 2.3 (left) provides an overview of the approach. As shown in the figure, the main idea in achieving asynchronous progress is to trigger an event once a control message arrives at a process. This interrupt invokes a callback handler which

processes the message and makes progress on the rendezvous. The required control messages which triggers the events in the existing scheme are: a) RNDV_START and b) RNDV_FINISH. In addition, the RDMA read completion also triggers a local completion event. This design provides good ability to overlap computation and communication via asynchronous progress. For example, if an application is busy doing computation, the callback handler can make progress via the interrupt mechanism. However, there are a couple of important details that arise in implementing the approach.

One main issue in the existing approach is the overhead of interrupt generation. As explained above, a total of three interrupts are generated for every rendezvous transfer of data. This can potentially degrade the performance for medium messages using this protocol. Further, it is not easy to provide for a mechanism to selectively ignore the events generated by the control messages. This feature can be used whenever the main application thread is already making progress and is expecting the control messages. Another important issue which cannot be overlooked is the overhead of locking/unlocking shared data structures. In this work, we take into account all these issues and propose a new implementation alternative. Specifically, we aim to:

- Avoid using locks for shared data structures
- Reduce the number of events triggered by the control messages
- Provide for an ability for the process to selectively ignore the events generated

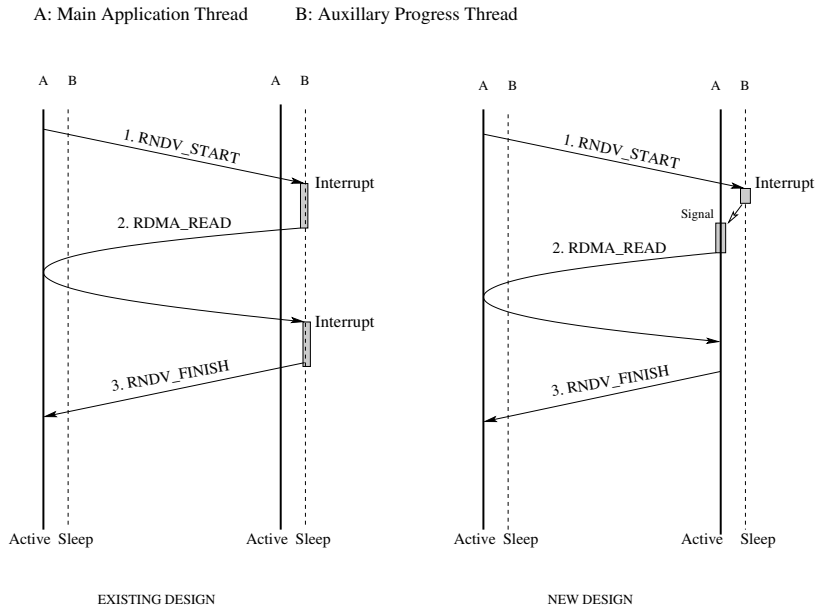


Figure 2.3: Asynchronous Rendezvous Protocol Implementations

2.3 Design of Asynchronous Progress

As explained above, the existing design has several limitations. In this Section, we explain our new approach of achieving asynchronous progress. Figure 2.3 (right) explains the basic idea in the new implementation. In our approach, each process creates an auxiliary thread at the beginning. The auxiliary thread waits for RNDV_START control message. As seen from the figure, the RNDV_START control message issued by the sender interrupts the auxiliary thread. This thread in turn sends a signal to the main thread to take the necessary action. This is different from the earlier approach where the auxiliary thread made progress on the rendezvous communication. Since, only one thread is involved with communication data structures, no locking mechanism is required for the data structures. In the second step, the main thread issues the RDMA read for the data transfer. After issuing RDMA read, the main thread

resumes to perform the computation. Unlike the existing approach, the RDMA read completion does not trigger any interrupt in our design. We believe this interrupt does not help in overlap in Single Program Multiple Data (SPMD) programming model where each process performs the same task and the load is equally balanced. Triggering of the interrupt on RDMA read completion can be easily added to the protocol if required. In our design, the main thread sends the RNDV_FINISH message soon after it discovers the completion of RDMA read.

There are several benefits of this new design. First, locks are avoided thus reducing contention for shared resources. Also, in our design the signal from the auxiliary thread is disabled by the main thread when it is not expecting a message from any process. By doing so, the main thread is not unnecessarily interrupted by an unexpected message since it does not have the receive buffer address to make progress on the communication. The main thread also disables signal if it is already inside the MPI library and making communication progress. Since the main thread can disable the interruption from the auxiliary thread, the execution time of the application is unaffected if rendezvous protocol is not used by the application. Also, the signal is enabled only if a non-blocking receive has been posted and not for blocking receives. Also, at most of the time the auxiliary thread is waiting for interrupts from the NIC and does not perform any communication processing. Therefore, as the auxiliary thread is I/O bound the dynamic priority of the thread is very high which helps in scheduling it quickly. Finally, the new design also cuts down the number of interrupts to one thus improving the communication performance.

2.4 Evaluation

The experiments were conducted on a 64 node InfiniBand Linux cluster. Each machine has dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of eight cores per node. Each node is connected by DDR network interface card MT25208 dual-port Memfree HCA by Mellanox [4] through a switch. InfiniBand software support is provided through OpenFabrics/Gen2 stack [23], OpenFabrics Enterprise Edition 1.2.

2.4.1 Comparison with existing design

Figure 2.4 shows the performance of basic bandwidth micro-benchmark. We used OSU Benchmarks [25] for the experiment. The legend 'no-async' refers to the basic RDMA read based rendezvous protocol without any enhancements for asynchronous progress, 'existing-async' refers to the existing asynchronous progress design proposed in [34] and 'new-async' refers to the proposed design described in Section 2.3. Figure

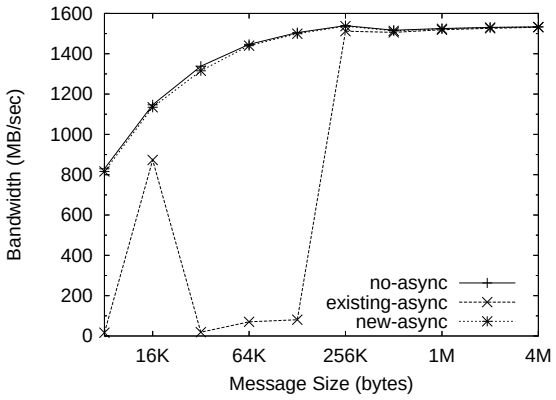


Figure 2.4: Bandwidth for large messages

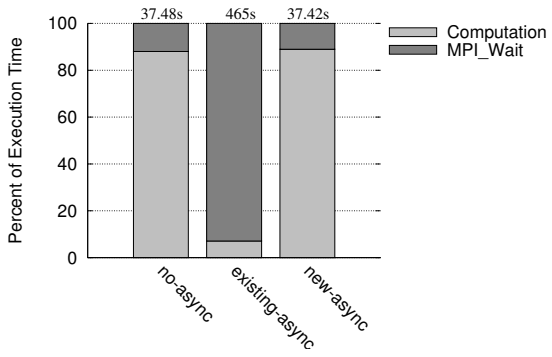


Figure 2.5: NAS-SP Normalized Execution Time

2.4 shows that the bandwidth of the proposed design closely matches with the base bandwidth numbers, which matches our expectations. However, with the old design

the bandwidth is very low. In the bandwidth test, the receiver posts several requests and waits for the completion of all the pending messages. As several rendezvous start messages are received by the process, the auxiliary thread is continuously interrupted. Also, since the main thread is not involved in computation, both the threads concurrently poll the MPI library. The main thread cannot make any progress, however, it hinders the auxiliary thread from being scheduled on the processor. Therefore, due to exhaustion of CPU resources by the main thread the bandwidth performance is affected. The bandwidth performance is also non-deterministic as it depends on the scheduler to schedule the auxiliary process quickly. The effects of schedule is discussed in Section 2.4.2.

The performance of the new design is very similar to the base bandwidth performance since the main thread disables interrupts from the auxiliary thread when it is already inside the MPI library.

The poor performance of the existing design can be seen not only on micro-benchmarks but also in the performance of SP NAS Parallel Benchmark [20] application as can be seen in Figure 2.5. It can be seen from the figure that with the old design most of the execution time is wasted in MPI_Wait. In the remaining evaluations we do not show the performance of the old design. We found that the old design performs well when using an extra-core, however, it performs poorly when a single processor is assigned per process.

2.4.2 Overlap Microbenchmark Performance

Figures 2.6 and 2.7 show the overlap performance of the proposed design. Sandia Benchmark [31] (SMB) has been used to evaluate the overlap capability of the implementation. Overlap potential at the receiver and at the sender have been shown in Figures 2.6 and 2.7, respectively. Since the base design and the proposed design

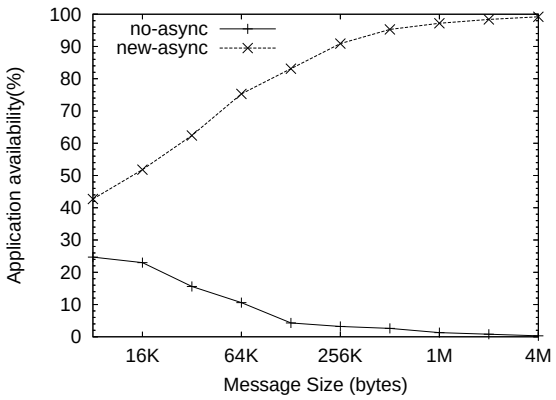


Figure 2.6: Application availability at Receiver

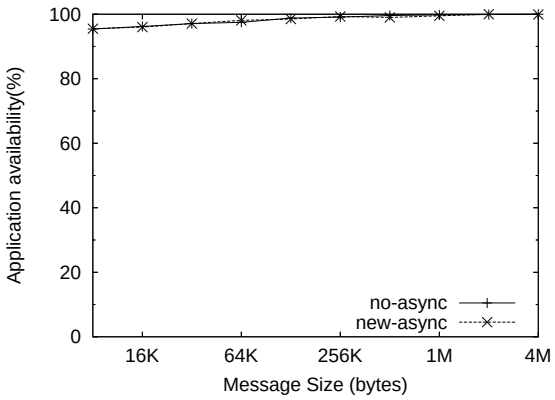


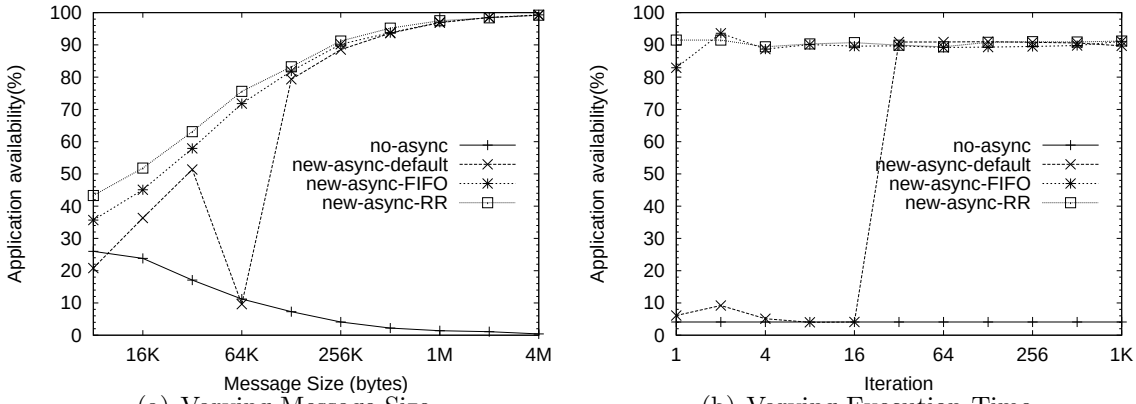
Figure 2.7: Application availability at Sender

employ RDMA read, almost total overlap is achieved at sender for both protocols. However, at the receiver the base RDMA read based protocol offers no overlap, as expected. The proposed design is able to achieve increasing overlap with increasing message size and reaches almost 100% overlap for messages greater than 1MB.

Effect of Schedule

Linux has different scheduling policies as discussed in appendix A. There are three scheduling policies FIFO (First in first out) , RR (Round-robin) and OTHER (the default policy). There is another policy BATCH which is used in batch systems and

has not been used in our evaluations. Figures 2.8(a) and 2.8(b) show the effect of scheduling algorithm on the overlap performance of the new design. Results for the default Linux schedule, FIFO and Round robin have been compared. For each of the executions with different scheduling algorithm, the auxiliary thread is assigned the highest possible priority so that it is scheduled as soon as it is interrupted.



(a) Varying Message Size (b) Varying Execution Time

Figure 2.8: Effect of scheduling algorithm

Figure 2.8(a) shows the results for different message sizes. We observe that with the default scheduling algorithm, the performance is not consistent for all message sizes. At some message sizes the auxiliary thread is not scheduled on the processor on being interrupted. However, with FIFO scheduling algorithm the performance improves and is best for round-robin algorithm. These real-time scheduling policies could not be used in the original design because it causes deadlocks. The deadlocks happen since real-time scheduling policies are non-premptive and the auxiliary thread

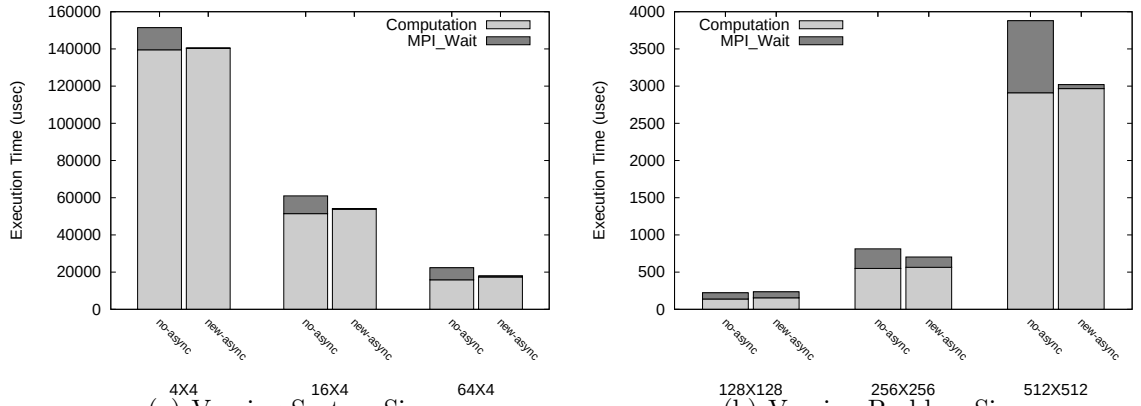
keeps spinning on spin-locks. Spin-locks are generally used to protect data in MPI implementation to obtain high performance.

Figure 2.8(b) shows the overlap performance for 256KBytes message with increased number of iterations in each execution. From the figure, it is observed that with the default scheduling algorithm the performance of the design improves after a certain time interval. We feel that the improved performance is due to the dynamic priority scheme of Linux scheduling algorithm. Since the auxiliary thread hardly uses the CPU and is mostly waiting for completion events it is assigned a high dynamic priority which helps increase its performance. However, for FIFO and round robin the performance is optimal even for low number of iterations.

2.4.3 Application Performance

In this Section, we use a matrix multiplication kernel to evaluate the application performance of the proposed design. The kernel uses Cannon's algorithm [15] and employs both MPI and OpenMP [24] programming models. The kernel requires the number of processes to be a perfect square. Since we wanted to use all 64 nodes of our cluster we could only use 4 cores per node in our experiments. However, since each thread is affined to a single core, the presence of the remaining unused cores of the nodes does not improve or affect the performance of the design. OpenMP programming model is used within the node and MPI is used for inter-node communication.

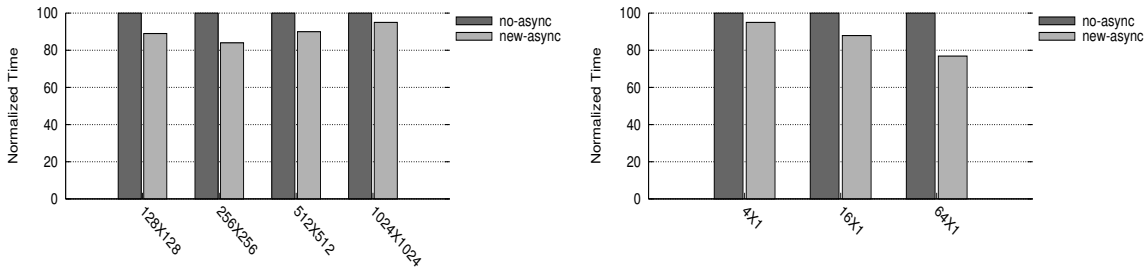
Figure 2.9(a) shows the application performance with increasing system sizes for a square matrix of dimensions 2048 elements. Each element of the matrix is a double datatype occupying eight bytes. As can be seen from the figure, the MPI_Wait time can be reduced by using the proposed design. Figure 2.9(b) shows the performance



(a) Varying System Size (b) Varying Problem Size

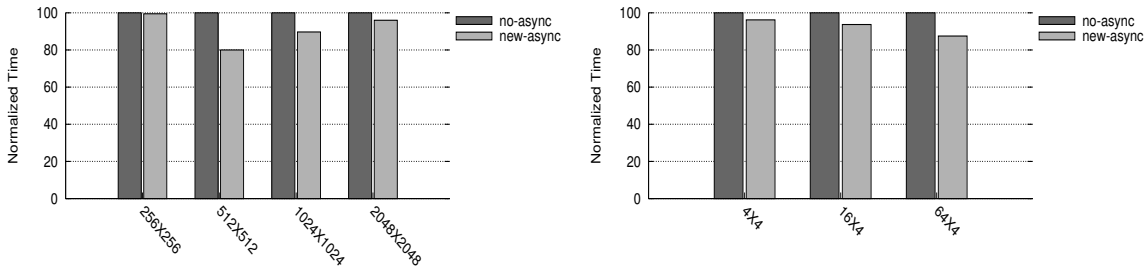
Figure 2.9: Matrix Multiplication: MPI + OpenMP configuration

for increasing problem size on four nodes and dividing the work of each node among four of its cores using OpenMP. Reductions in MPI_Wait time can also be seen with different problem sizes. For matrix of 128X128 dimensions, no improvement is observed as the message communication is of size 4K Bytes which does not employ rendezvous protocol.



(a) Varying System Size (b) Varying Problem Size

Figure 2.10: Matrix Multiplication: MPI x1 configuration



(a) Varying System Size

(b) Varying Problem Size

Figure 2.11: Matrix Multiplication: MPI x4 configuration

The performance of the new design has also been evaluated for MPI programming model without using OpenMP. The MPI implementation of matrix multiplication kernel was run in two configurations. In the first configuration, all the processes are involved in inter-node communication. This is achieved by launching only one process per node. In the second configuration, four processes per node are launched. In this setup, some of the processes employ shared memory for communication whereas some processes are involved in inter-node communication. So the processes which use shared memory communication cannot achieve any overlap. The results for the first configuration can be seen in Figures 2.10(a) and 2.10(b). The results for the second configuration can be seen in Figures 2.11(a) and 2.11(b). In Figure 2.11(a), no improvement is observed for matrix of dimensions 256X256. This is because the size of message transfer is less than 8KB. Messages of sizes lower than 8KB were not using rendezvous protocol in the experiments. Considerable improvement is observed for all the other problem sizes and system sizes. However, the improvement in performance is lower than when MPI+OpenMP programming model is used. This is because of

decreased percentage of communication time (of the corresponding execution time) than the MPI+OpenMP program.

2.5 Summary

In this chapter, we proposed an enhanced asynchronous rendezvous protocol. Specifically, our design does not require locks in the communication path. In our approach, the main application thread makes progress on the rendezvous transfer with the help of an additional thread. The communication between the two threads occurs via system signals. The new design achieves almost total overlap of communication with computation. Further, our design does not reduce the performance of non-overlapped communication. With the new design we have been able to achieve 20% reduction in time for a matrix multiplication kernel with MPI+OpenMP paradigm on 256 cores.

CHAPTER 3

ALLTOALL COLLECTIVE FOR ONLOAD AND OFFLOAD INTERFACES

Alltoall collective operation is extensively used in many applications which includes CPMD [3], NAMD [26], FFT and matrix transpose. Alltoall is a very communication intensive operation and scales poorly on large systems. Therefore, the scalability of applications, which involve this communication pattern, is limited by the implementation of this collective. There has been tremendous research in optimizing this collective for SMP and uni-processor systems. In this Chapter, we take on the challenge of designing communication algorithm of alltoall collective for multi-core systems connected with InfiniBand interfaces. We study the characteristics of both onload and offload InfiniBand interfaces and evaluate algorithms on both architectures. We evaluate our designs on clusters connected with InfiniPath, InfiniHost III and ConnectX InfiniBand network interfaces.

The rest of the chapter is organized as follows. In Section 3.1, we provide the necessary background information for this work. In Section 3.2, we provide the motivation behind the work. In Section 3.3, we present the related work. In Section 3.4, we discuss the proposed designs. In Section 3.5, we evaluate our design and provide experimental results. Finally, in Section 3.6, we summarize the results of this work.

3.1 Background

In this Section, we provide the necessary background details for this work. First, we describe the current alltoall algorithms employed in popular MPI implementations. Then we describe CPMD [3] application. CPMD has been used in the evaluation of our design.

3.1.1 Alltoall Algorithms

The most popular algorithms for alltoall currently used are: 1. Bruck's algorithm [7], 2. Irecv-Isend algorithm [35] and 3. Pairwise Exchange [35].

Because none of the above algorithms gives the best performance for all message sizes, we choose different algorithms according to the message size. Bruck's algorithm completes in minimum number of steps, log P (P is the number of processes). Hence, it is used for small messages where start-up latencies are a dominant part of the collective time. However, because it sends the same message over the network more than once, it is not suitable for medium or large messages. In the Irecv-Isend algorithm, each process sends the data directly to the destined process, hence it requires P-1 steps to complete. The amount of data going out of each node is equal to the total amount of data that each node must send. The amount of data going into each node is equal to the total amount of the data that each node must receive. Therefore, the algorithm is optimal in terms of amount of data sent on the network and should be suitable for medium and large messages. However, we found the algorithm is not suitable for large messages. At large message sizes, contention on the links comes into play. The algorithm uses a cyclic pattern of communication which is not congestion free on fat-tree networks [14]. The pairwise exchange algorithm gives better results for

large messages. At each stage of the pair-wise exchange algorithm, the communication pattern is congestion free on fat-tree networks. Moreover, 'irecv-isend' algorithm makes loose coupling among the sending and receiving processes. It has been found that if processes are tightly coupled, the latencies are lower for large messages [30]. Pair-wise exchange uses send-recv, utilizing rendezvous protocol for large messages and hence are tightly coupled.

We have found that network characteristics play a role in tuning the alltoall collective for different network interfaces. For example, the 'irecv-isend' algorithm performs poorly on InfiniHost III and ConnectX. However, it performs well for medium-sized messages on the InfiniPath network interface. All of the above tuning are available in the open source MVAPICH [21] software.

3.1.2 CPMD

The Car-Parrinello Molecular Dynamics (CPMD) [3] is designed for ab-initio molecular dynamics. It is widely used for research in computational chemistry, materials science and biology. It was developed by IBM Research Zurich laboratory and the Max-Planck-Institute, Stuttgart in collaboration with many groups around the world. It is used world wide by more than 6000 users. The application is a production code mainly written in FORTRAN, parallelized for distributed-memory with MPI. CPMD makes extensive use of three-dimensional FFT, which requires efficient all-to-all communication [12]. CPMD application has been used in the evaluation of the proposed design.

3.2 Motivation

Communication time of MPI_Alltoall is dependent on two factors: start-up costs and network bandwidth. For small messages, MPI_Alltoall time is dominated by start-up costs. For large messages, network bandwidth determines the time of the operation. To illustrate the impact of start-up costs on latency of MPI_Alltoall, we conducted a simple test to measure the time of 'irecv-isend' alltoall algorithm on a fixed set of nodes. However, we increase the number of cores involved in the collective keeping the size of total data involved in the operation the same. The experiment was conducted on our InfiniPath cluster mentioned in section 3.5. Figure 3.1 shows the results. As shown in the figure, we observe that there is a significant increase in

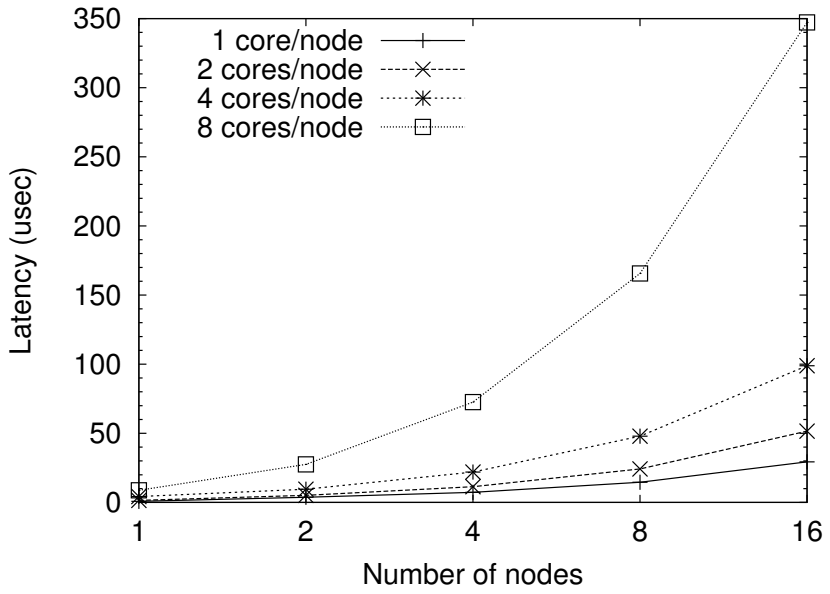


Figure 3.1: Performance of alltoall on multi-core systems

MPI_Alltoall time with the increase in the number of cores per node although the

amount of data exchanged between the nodes is the same. This is primarily due to an increasing number of sends issued which increases start-up costs. Thus, reducing start-up costs is necessary to obtain good performance.

3.2.1 Performance of bi-directional bandwidth

On the other hand, different implementations of InfiniBand network interfaces exist. These different interfaces exhibit varying communication characteristics. We demonstrate this using a simple bi-directional bandwidth test between two nodes. The number of concurrent pairs involved in the test is increased from one to four. Figures 3.2 and 3.3 show the multi-pair bi-directional bandwidth performance on InfiniPath and ConnectX adapters, respectively.

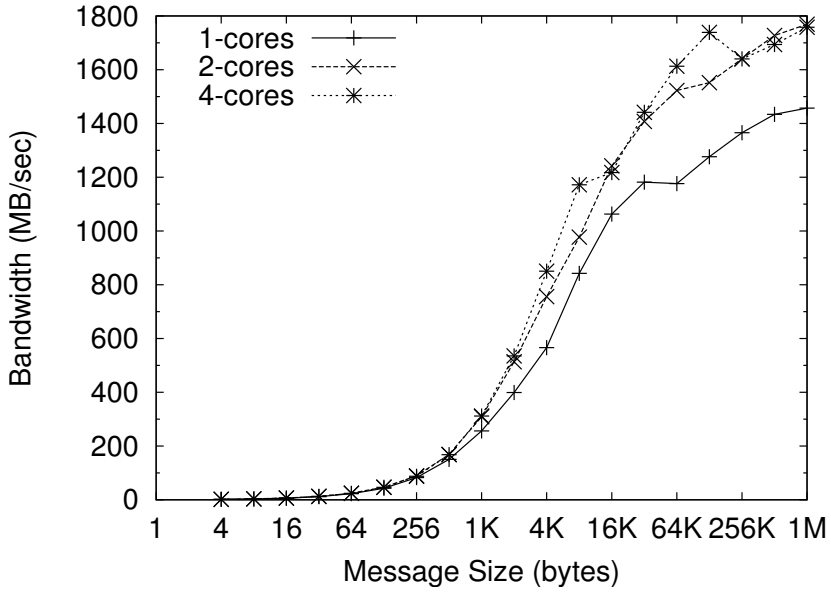


Figure 3.2: InfiniPath SDR: Multi-pair Bidirectional Bandwidth.

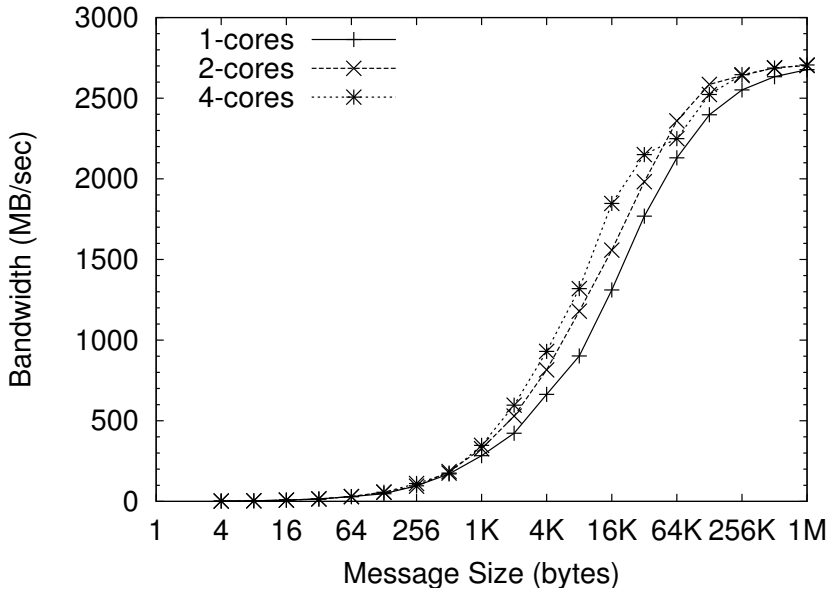


Figure 3.3: ConnectX DDR: Multi-pair Bidirectional Bandwidth.

We observe that using more than one core to send the data out of the node is advantageous as it provides better network utilization, as can be seen by the increase in bandwidth on using more cores. This is not the case for earlier generation InfiniHost III architecture, as can be seen in Figure 3.4. For MPI_Alltoall operations, this observed behavior is significant because, like the bi-directional test, multiple cores are involved in data exchange across the nodes.

Thus, as described above, different network interfaces exhibit varying communication characteristics. The alltoall schemes need to take into account these factors to obtain good performance.

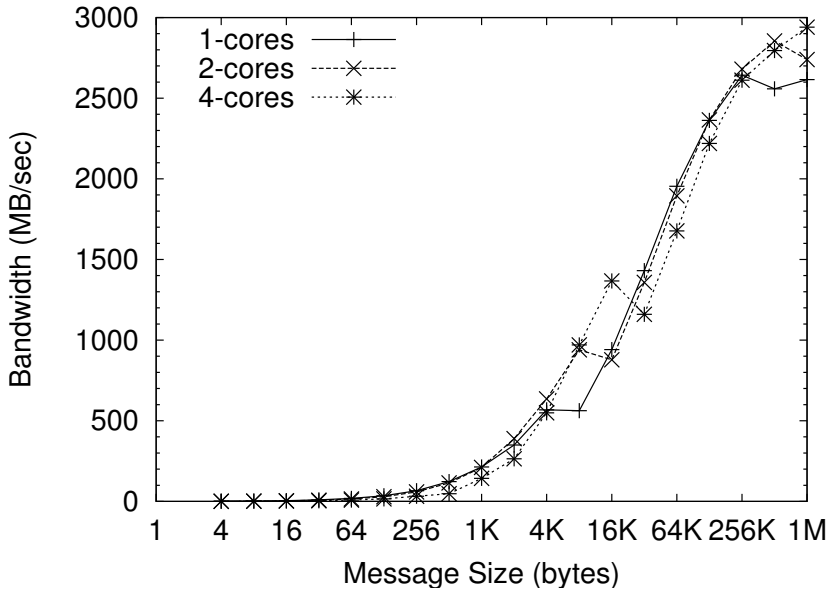


Figure 3.4: InfiniHost III DDR: Multi-pair Bidirectional Bandwidth.

3.3 Related Work

Several optimizations have been proposed in the past to take advantage of shared memory to design collectives. Husbands et al. [11] first proposed hierarchical tree based MPI_Bcast algorithm to minimize the use of network on the Sun SMP system. Sistare et al. [32] propose a hierarchical scheme but do not use a tree-based point to point communication within the SMP node. They develop shared memory based schemes to optimize broadcast, reduce, allreduce and barrier within the SMP node. Tipparaju et al. [36] also propose hierarchical tree based collective operations using shared and remote memory access protocols. In an earlier work from our group [17], we proposed a hierarchical multicast based design for broadcast. Most of the work to optimize collectives for shared memory based systems have proposed hierarchical

leader based schemes. In this work we propose a non-leader based scheme to optimize alltoall collective for multi-core systems.

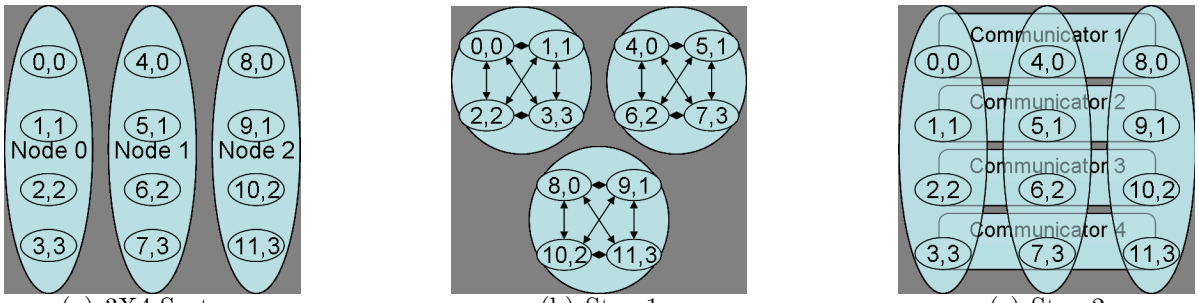
3.4 Proposed Design

In the leader based scheme, at the sender, all data of a node is aggregated to the leader of the node followed by inter-node communication and then distribution to all the processes of the node at the receiver. The leader based scheme proposed for SMP based clusters cannot be naively used for all multi-core systems. It has the following disadvantages which are addressed by the proposed design:

1. Significant shared memory overhead (assuming all the intra-node communication is done using shared memory communication). This is because the inter-node alltoall can begin only after all processes of the node have written data to the shared memory location.

2. Utilizes a single core to perform the inter-node communication and therefore does not take advantage of the increase in bi-directional bandwidth available with the increase in number of cores used to send the data.

One can also design a leader-based scheme with two leaders per node. This scheme would achieve better bandwidth as it utilizes more cores to send the data to other nodes. However, this also increases the number of network sends by two times and hence increases start-up costs. Instead, a scheme in which more cores of the node participate in inter-node communication without an increase in number of sends issued by each core will benefit significantly. This can be achieved if each core of a node communicates with one and only one core of all other nodes. This does not increase



(a) 3X4 System (b) Step 1 (c) Step 2

Figure 3.5: Communication steps of the proposed design

the number of network sends issued by each process and network start-up costs are almost the same with better network utilization. In our implementation of the above scheme, we opted to use all cores of a node to participate in inter-node communication. However, a subset of the cores can also be used. We explain the design keeping in mind that all of the cores are being used. However, it can be extended to address the situation wherein only a subset of the cores are utilized.

Since a core/process communicates with a single core of the other nodes, intra-node communication must be used to send the respective data to the other cores of the node. There are two ways in which this can be performed: 1. Before sending the data to other nodes, perform an intra-node communication. In this, a core receives all data that has to be received by cores with which it will communicate. We call this send-side aggregation. OR 2. Perform the inter-node communication first. In the inter-node communication, a core sends all of its data destined for a node to the core with which it communicates. It then performs an intra-node communication in

which a core sends the data to the respective cores for which it was destined. We call this receive-side distribution.

We chose the first option for our implementation.

The proposed algorithm for alltoall completes in two steps:

1. Step 1: Intra-node exchange - This step takes place simultaneously within all nodes. Each core/process sends all the data that has to go to shared memory rank x (rank of the process in its node) of all nodes to process with shared memory rank x on its node.

2. Step 2: Inter-node exchange - Each core/process performs an inter-node alltoall communication with processes having the same shared memory rank. The message size of this alltoall is more than that of the intended alltoall communication.

Figure 3.5 shows the communication that takes place in each of the above steps for a system of size 3X4, i.e., the system having three nodes and each node has four cores as shown in Figure 3.5(a). Each small ellipse represents a core and the surrounding bigger ellipse represents a node. The numbers 'a' and 'b' on the core represent 'a' as MPI_Rank and 'b' as the shared memory rank. Figure 3.5(b) shows the communication of step 1. In this step, only intra-node communication takes place. After step 1, all processes having the same shared memory rank are part of one communicator as shown in Figure 3.5(c). Each of the core/process participates in alltoall communication within the new communicator.

The above scheme has the following advantages:

1. Lower shared memory overhead. Each process waits for other processes to write only a subset of their data not their whole data.

2. Uses more than one core to send out the data, allowing for better bandwidth.

3.5 Evaluation

The following two test-beds were used to conduct the experiments:

First is a 512-core InfiniBand Linux cluster. Each of the 64 nodes have dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of 8 cores per node. Each node has two network interface cards. First is a host-based SDR network interface QLE7140 by Qlogic and the second is offload DDR network interface card MT25208 dual-port Memfree HCA by Mellanox. InfiniBand software support is provided through InfiniPath software stack 2.1 on Qlogic HCA and OpenFabrics/Gen2 stack [23], OFED 1.2 release for Mellanox HCA. The Mellanox HCA is built using the Infinihost III architecture.

Second is a 4 node dual 2.33 GHz Intel Xeon "Clovertown" quad-core processors for a total of 8 cores per node. Each node is connected with Mellanox ConnectX cards which operate at DDR speed (20Gbps). The ConnectX card (MT25408) has firmware version 2.0.139 and operates with new Open-Fabrics drivers which are based on OFED 1.2 distribution.

We have used MVAPICH-PSM and MVAPICH-Gen2 [21] to test our collective schemes on the two devices. MVAPICH is a popular open-source MPI implementation over InfiniBand. It is based on MPICH [10] and MVICH [16] and is used by over 715 organizations worldwide.

3.5.1 Alltoall Performance

In this section, we evaluate and compare the performance of the proposed scheme using OSU Alltoall Benchmark [25]. The benchmark calls MPI_Alltoall back-to-back and reports an average over a large number of iterations (typically 1000).

Performance on InfiniPath

Figure 3.6 shows the MPI_Alltoall collective performance on 64X8 configuration where AXB implies 'A' nodes and 'B' cores per node. The legend 'orig' refers to the current algorithms employed in MVAPICH tuned for the testbed for appropriate message sizes.

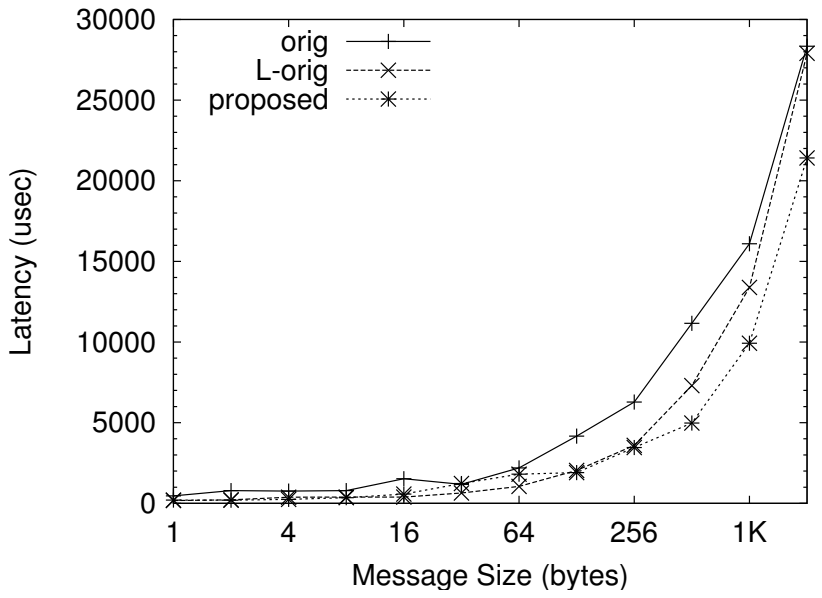


Figure 3.6: InfiniPath: Alltoall time on 64X8 system

MVAPICH-PSM currently uses Bruck's algorithm for up to 256 Bytes, direct 'irecv-isend' from 256Bytes to 32KB and pairwise exchange for messages larger than

32KB for the Qlogic HCA. We have found that the direct 'irecv-isend' algorithm performs poorly on Mellanox HCA; therefore MVAPICH-GEN2 uses Bruck's algorithm for up to 8KB and pairwise exchange for messages larger than 8KB. The 'L-orig' scheme refers to leader based scheme and uses the original tuned alltoall to perform the inter-node alltoall communication among the leaders. The 'proposed' scheme refers to the new proposed scheme explained in section 3.4. It uses the tuned alltoall explained above to perform step 2 of the proposed scheme.

The leader based scheme performs well for very small messages. However, due to high shared memory overhead, the benefits fade with increasing message size. The proposed scheme outperforms the current algorithm and leader-based algorithm up to 2KB message size. This is primarily due to better utilization of network bandwidth by using multiple cores. As the system size increases, higher performance gains are obtained and up to a greater message size.

Figure 3.7 shows the MPI_Alltoall time for 512Byte message on varying system sizes. The results show that the performance gains in alltoall time increase with increasing system sizes.

Performance on Infinihost III

On Infinihost NIC, simultaneously using multiple cores deteriorates the performance of communication latency [33]; therefore, multi-pair bi-directional bandwidth shows deterioration in performance with increasing number of cores as can be seen in Figure 3.4. Therefore, leader-based scheme performs best here because it is able to eliminate the effects of congestion. This can be seen from the results in Figure 3.8.

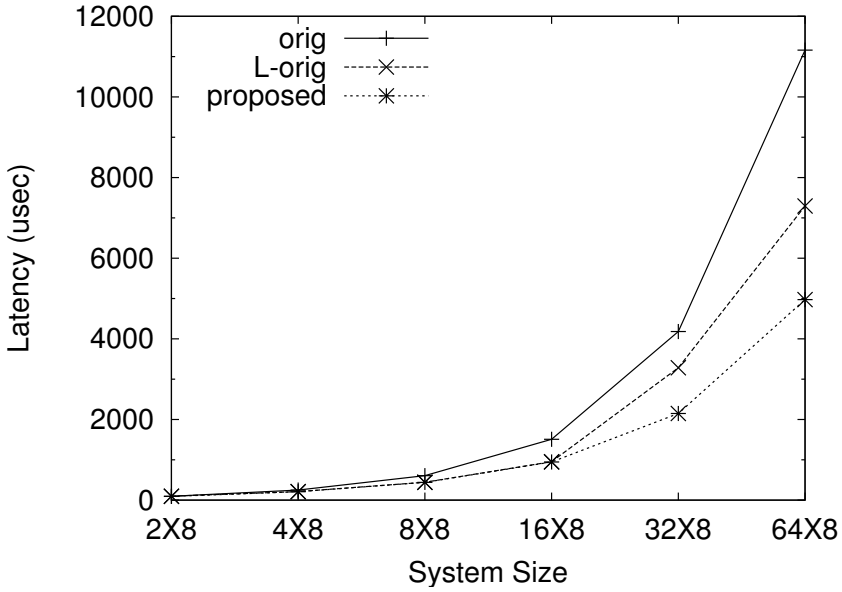


Figure 3.7: InfiniPath: Alltoall time of 512Byte message

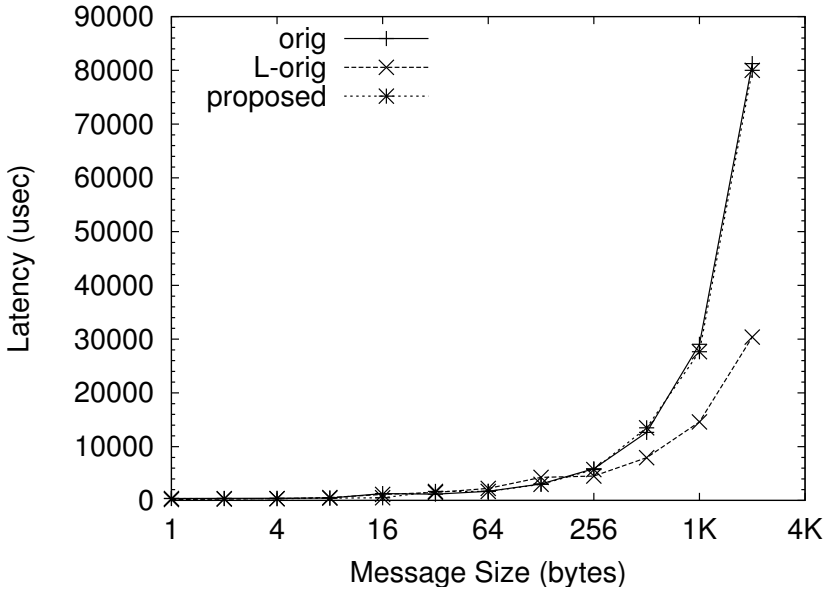


Figure 3.8: Infinihost III: Performance of different schemes on 64X8 system

Performance on ConnectX

From Figure 3.3 we see that on ConnectX architecture, multi-pair bi-directional bandwidth increases with more cores. Therefore, leader-based scheme does not perform as well as the proposed scheme as it uses only a single core for network communication. However, both schemes perform better than the original scheme as they reduce the number of network transmissions. The alltoall time for different schemes can be seen in Figure 3.9.

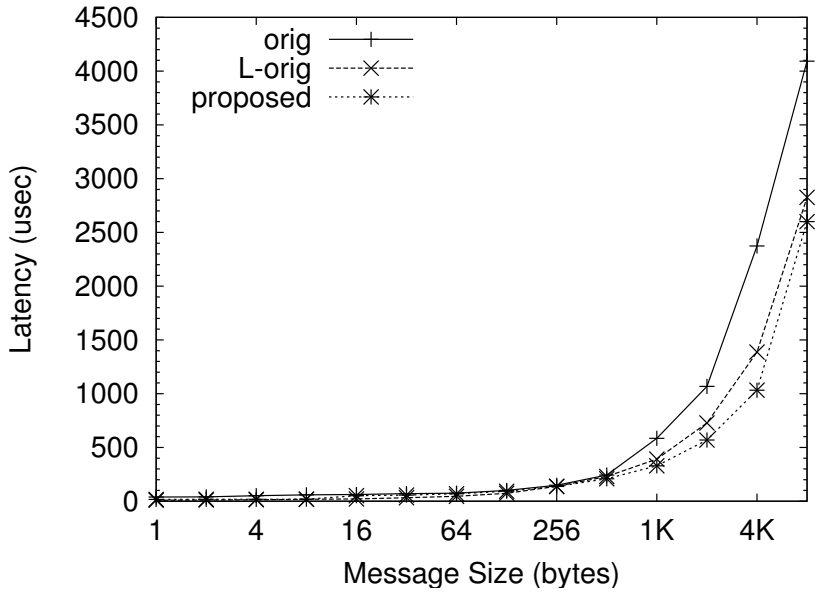


Figure 3.9: ConnectX: Performance of different schemes on 4X8 system

3.5.2 Application Performance

The CPMD application was used to evaluate the performance impact of the proposed scheme on applications. The InfiniPath network interface testbed was used

for the evaluation. Figure 3.10 shows the performance improvement over the current

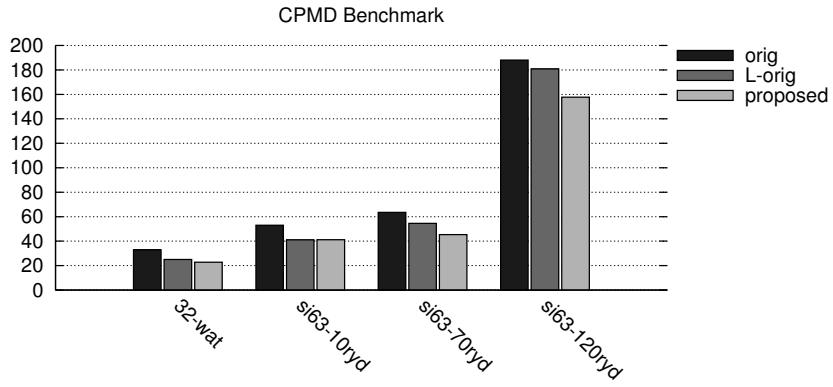


Figure 3.10: CPMD Application Benchmark Performance on InfiniPath: Different Input Files

algorithms for different input files on 16X8 system.

Figure 3.11 shows the performance improvement for si63 atoms with 120ryd cut-off for different system sizes. As we saw earlier, the performance improvement for MPI_Alltoall increases with increasing system sizes, this is also reflected in the application performance improvement. Figure 3.11 shows that the 'L-orig' scheme begins to perform well at 64X8 system size. At 64X8 system size, the message size of alltoall collective decreases as the problem size remains the same. Therefore, leader-based collective performs comparable to the proposed scheme.

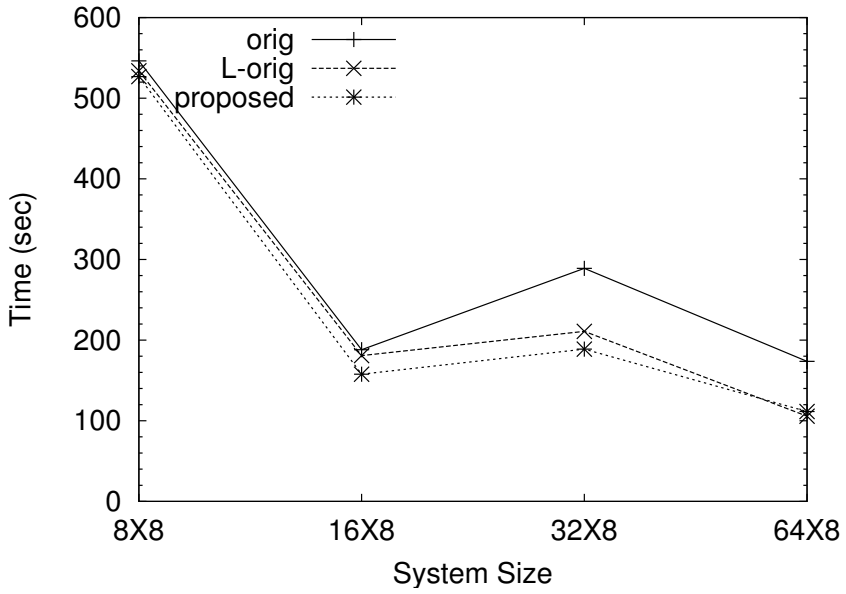


Figure 3.11: CPMD Application Benchmark Performance on InfiniPath: Varying System Sizes

3.6 Summary

In this chapter, we have extended the leader-based collective design to implement the alltoall collective for multi-core systems. We have also proposed a partial aggregation based design for alltoall collective. We evaluated both the designs in detail. The results show that the performance of the designs differ on onload and offload interfaces. The leader-based design is more suitable for offloaded interfaces such as InfiniHost III architecture. Whereas, the partial aggregation based design produces better performance on onload interfaces such as InfiniPath architecture.

A single collective algorithm does not perform optimally for all network interfaces due to differing network characteristics. Our results show that the proposed scheme delivers improvement in the performance of CPMD application by as much as 33% .

CHAPTER 4

CONTRIBUTIONS AND FUTURE WORK

In this thesis, we have studied the characteristics of InfiniBand interfaces when the network processing is offloaded to the NIC and when it is performed by the host processor. We describe how we can take advantage of the features provided by such interfaces. Our work involved designing collective algorithms and enhancement to point-to-point communication protocols.

4.1 Summary of Research Contributions and Future Work

The research in this thesis aims towards enhancing MPI by taking advantage of the features provided by offload and onload InfiniBand interfaces. The designs have been integrated into the MVAPICH software and are freely available for download. Following is a detailed summary of the research presented in the thesis.

4.1.1 Providing Asynchronous Progress in Rendezvous Protocol

There are several designs that have been proposed in the past to provide asynchronous progress. These designs typically use progress helper threads with support from the network hardware to make progress on the communication. However, most

of these designs use locking to protect the shared data structures in the critical communication path. Secondly, multiple interrupts may be necessary to make progress. Further, there is no mechanism to selectively ignore the events generated during communication.

In this thesis, we proposed an enhanced asynchronous rendezvous protocol which overcomes these limitations. Specifically, our design does not require locks in the communication path. In our approach, the main application thread makes progress on the rendezvous transfer with the help of an additional thread. The communication between the two threads occurs via system signals. The new design achieves almost total overlap of communication with computation. Further, our design does not reduce the performance of non-overlapped communication. With the new design we have been able to achieve 20% reduction in time for a matrix multiplication kernel with MPI+OpenMP paradigm on 256 cores. In future, we plan to carry out scalability studies of this new design for a range of applications and system sizes.

4.1.2 Improving Performance of Alltoall Collective

In Chapter 2, we studied the network characteristics exhibited by offloading/onloading of network processing in InfiniBand network interfaces. The results show that network interfaces implemented for the same interconnect exhibit different network characteristics. A single collective algorithm does not perform optimally for all network interfaces due to differing network characteristics. The thesis proposes an optimized alltoall collective algorithm for multi-core systems connected using modern InfiniBand network interfaces. However, we believe that the work can be applied to onload implementation of other networks as well, like the Ethernet-based JNIC architecture [9].

We plan to evaluate our designs on such systems in the future, as well as, extend the proposed framework to other collectives.

APPENDIX A

SCHEDULING POLICIES

The scheduler is the kernel part that decides which runnable process will be executed by the CPU next. The Linux scheduler offers three different scheduling policies, one for normal processes and two for real time applications. A static priority value sched_priority is assigned to each process and this value can be changed only via system calls. Conceptually, the scheduler maintains a list of runnable processes for each possible sched_priority value, and sched_priority can have a value in the range 0 to 99. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines for each process, where it will be inserted into the list of processes with equal static priority and how it will move inside this list. Currently, the following three scheduling policies are supported under Linux: SCHED_FIFO, SCHED_RR, SCHED_OTHER, and SCHED_BATCH; their respective semantics are described below.

SCHED_OTHER is the default universal time-sharing scheduler policy used by most processes. SCHED_BATCH is intended for "batch" style execution of processes. SCHED_FIFO and SCHED_RR are intended for special time-critical applications that

need precise control over the way in which runnable processes are selected for execution.

Processes scheduled with SCHED_OTHER or SCHED_BATCH must be assigned the static priority 0. Processes scheduled under SCHED_FIFO or SCHED_RR can have a static priority in the range 1 to 99. The system calls sched_get_priority_min() and sched_get_priority_max() can be used to find out the valid priority range for a scheduling policy in a portable way on all POSIX.1-2001 conforming systems.

All scheduling is preemptive: If a process with a higher static priority gets ready to run, the current process will be preempted and returned into its wait list. The scheduling policy only determines the ordering within the list of runnable processes with equal static priority.

SCHED_FIFO: First In-First Out scheduling SCHED_FIFO can only be used with static priorities higher than 0, which means that when a SCHED_FIFO processes becomes runnable, it will always immediately preempt any currently running SCHED_OTHER or SCHED_BATCH process. SCHED_FIFO is a simple scheduling algorithm without time slicing. For processes scheduled under the SCHED_FIFO policy, the following rules are applied: A SCHED_FIFO process that has been preempted by another process of higher priority will stay at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again. When a SCHED_FIFO process becomes runnable, it will be inserted at the end of the list for its priority. A call to sched_setscheduler() or sched_setparam() will put the SCHED_FIFO (or SCHED_RR) process identified by pid at the start of the list if it was runnable. As a consequence, it may preempt the currently running process if it has the same priority. A process calling sched_yield() will be put at the end of the

list. No other events will move a process scheduled under the SCHED_FIFO policy in the wait list of runnable processes with equal static priority. A SCHED_FIFO process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, or it calls sched_yield().

SCHED_RR: Round Robin scheduling SCHED_RR is a simple enhancement of SCHED_FIFO. Everything described above for SCHED_FIFO also applies to SCHED_RR, except that each process is only allowed to run for a maximum time quantum. If a SCHED_RR process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A SCHED_RR process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum. The length of the time quantum can be retrieved using sched_rr_get_interval.

SCHED_OTHER: Default Linux time-sharing scheduling SCHED_OTHER can only be used at static priority 0. SCHED_OTHER is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice level and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all SCHED_OTHER processes.

This text is based on linux man pages.

BIBLIOGRAPHY

[1] http://www.lanl.gov/roadrunner/.

[2] http://www.mellanox.com/products/infinihost_iii_ex_cards.php.

[3] http://www.cpmd.org/.

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

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

[6] Ron Brightwell and Keith D. Underwood. An Analysis of the Impact of MPI Overlap and Independent Progress. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, March 2004.

[7] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions in Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.

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

[9] Mike Schlansker et. al. High-performance Ethernet-based Communications for future Multi-core Processors. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 49–59, 2007.

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

[11] P. Husbands and J.C. Hoe. MPI-StarT: Delivering Network Performance to Numerical Applications. *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, pages 17–17, 07-13 Nov. 1998.

[12] J. Hutter and A. Curioni. Dual-level Parallelism for Ab Initio Molecular Dynamics: Reaching Teraflop Performance with the CPMD Code. In *Parallel Computing*, pages 1–17, 2005.

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

[14] Sameer Kumar and Laxmikant V. Kale. Scaling All-to-All Multicast on Fat-tree Networks. *ICPADS: International Conference on Parallel and Distributed Systems*, 00:205, 2004.

[15] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., 1994.

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

[17] A.R. Mamidala, Lei Chai, Hyun-Wook Jin, and D.K. Panda. Efficient SMP-aware MPI-level Broadcast over InfiniBand's Hardware Multicast. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, 25-29 April 2006.

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

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

[20] NAS Parallel Benchmarks (NPB). http://www.nas.nasa.gov/Software/NPB/.

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

[22] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. Thousand Oaks, CA, USA, 2006. Sage Publications, Inc.

[23] OpenFabrics Alliance. OpenFabrics. http://www.openfabrics.org/.

[24] OpenMP. http://openmp.org/wp/.

[25] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

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

[27] Qlogic. InfiniPath. http://www.pathscale.com/infinipath.php.

[28] Quadrics. QsNET II. http://www.quadrics.com/quadrics.

[29] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Huggahalli, Don Newell, Linda Cline, and Annie Foong. TCP Onloading for Data Center Servers. *Computer*, 37(11):48–58, 2004.

[30] D. Roweth and A. Moody. Performance of All-to-All on QsNetII, Quadrics White Paper, available at http://www.quadrics.com/. 2005.

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

[32] S. Sistare, R.v. Vaart, and E. Loh. Optimization of MPI Collectives on Clusters of Large-Scale SMPs. *Supercomputing, ACM/IEEE 1999 Conference*, pages 23–23, 13-18 Nov. 1999.

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

[34] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Symposium on Principles and Practice of Parallel Programming, (PPOPP '06)*, March 2006.

[35] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective communication operations in MPICH. *Int'l Journal of High Performance Computing Applications*, 19(1):49–66, Spring 2005.

[36] V. Tipparaju, J. Nieplocha, and D.K. Panda. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, 22-26 April 2003.