DESIGNING SUPPORT FOR MPI-2 PROGRAMMING INTERFACES ON MODERN INTERCONNECTS

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

Tejus Gangadharappa, B.E

* * * * *

The Ohio State University

2009

Master's Examination Committee:

Dr. D.K. Panda, Adviser

Dr. P. Sadayappan

Approved by

____

Adviser

Graduate Program in Computer Science and Engineering

© Copyright by

Tejus Gangadharappa

2009

ABSTRACT

Scientific computing has seen an unprecedented growth in the recent years. The growth of high performance interconnects and the emergence of multi-core processors have fueled this growth. Complement to the growing cluster sizes, researchers have developed varied parallel programming models to harness the power of larger clusters. Popular parallel programming models in use range from traditional message passing and shared memory models to newer partitioned global address space models. MPI, a de-facto programming model for distributed memory machines was extended in MPI-2 to support two new programming paradigms: the MPI-2 dynamic process management interface and the MPI-2 remote memory access interface.

The MPI-2 dynamic process management provides MPI applications the flexibility to dynamically alter the scale of the job by allowing applications to spawn new processes, making way for a master/slave paradigm in MPI. The MPI-2 remote memory access interface allows applications the illusion of globally accessible memory. In this thesis, we study the two MPI-2 programming interfaces and propose optimized designs for the them. We design a low overhead connection-less transport based dynamic process interface and demonstrate the effectiveness of our design using benchmarks. We address the design of the remote memory interface on onload-ed InfiniBand using a DMA copy offload. Our design of the remote memory interface provides for computation-copy overlap and minimal cache pollution. The proposed designs are

implemented and evaluated on InfiniBand, a modern interconnect which provides a rich set of features. The designs developed as a part of this thesis are available in MVAPICH2, a popular open-source implementation of MPI over InfiniBand used by over 900 organizations.

This is dedicated to my parents, friends and other random people

ACKNOWLEDGMENTS

The satisfaction that accompanies the successful completion of any task would be incomplete without the mention of those who made it possible.

I would like to thank my adviser, Prof. D. K. Panda for guiding me throughout the duration of my M.S. study. I appreciate the time and effort he invested in steering my work. I am thankful to Prof. P. Sadayappan for agreeing to serve on my Master's examination committee.

I acknowledge the students of the Network-Based Computing Laboratory (Nowlab), particularly Matthew Koop, Jaidev Sridhar, Gopal Santhanaraman, Karthik Gopalakrishnan and Wei Huang for their willingness to work with and help me on several occasions.

I would also like to thank all my Nowlab colleagues Hari Subramoni, Karthik B Gopalakrishnan, Krishan Kandalla, Xiangyong Ouyang, Ping Lai, Greg Marsh, Miao, Ajay and Sreeram. Finally, my acknowledgements to Jonathan for helping me in several system/cluster issues. Thanks to all the people who made the years at Ohio State fun, especially Jaidev, Deepak and Santosh.

Finally, I would like to thank my family members and friends back home.

VITA

June 14, 1981 .... Born - Bangalore, India

2003 .... B.E., Computer Science and Engg., Visvesvaraya Technological University, Belgaum, India.

2004-2005 .... Member of Technical Staff, Adobe Systems India

2005-2007 .... Software Development Engineer, Citrix R&D India

2008-2009 .... Graduate Research Associate, The Ohio State University

PUBLICATIONS

Research Publications

Tejus Gangadharappa, Matthew Koop and D. K. Panda "Designing and Implementing MPI-2 Dynamic Process Management On InfiniBand". *Workshop on Parallel Programming Models and Systems Software (P2S2), Held in conjunction with Int'l Conference on Parallel Processing (ICPP), Sept 2009. To be Presented.*

Tejus Gangadharappa, Gopal Santhanaraman, Karthik Gopalakrishnan, Sreeram Potluri and D. K. Panda "Improving MPI One-sided Passive Communication Using I/OAT Offload Engines". *OSU Technical Report. April 2009*

FIELDS OF STUDY

Major Field: Computer Science and Engineering

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

TABLE OF CONTENTS

Page

Abstract ii

Dedication iv

Acknowledgments v

Vita vi

List of Tables x

List of Figures xi

Chapters:

1. Introduction 1

    1.1 Overview of Modern Interconnects 2
    1.2 Overview of InfiniBand 3
        1.2.1 Communication Model 3
        1.2.2 Transport Modes 4
        1.2.3 Offloaded InfiniBand Interfaces 5
        1.2.4 Onloaded InfiniBand Interfaces 6
    1.3 Overview of Message Passing Interface (MPI) 7
        1.3.1 MPI Communicators 7
        1.3.2 Point-to-point communication 8
        1.3.3 Collective communication 9
    1.4 Problem Statement 9
    1.5 Organization of Thesis 10

2. Designing MPI-2 Dynamic Process Management 11

2.1 Background 12
2.1.1 MPI Inter-communicator 12
2.1.2 Dynamic Process API 13
2.2 Design of Dynamic Process Management 15
2.2.1 Dynamic process framework 15
2.2.2 Spawn Phase 16
2.2.3 Scheduling Phase 16
2.3 Design of Dynamic Process Management: Communication Phase . 17
2.3.1 Communication Methods 17
2.3.2 MPI_Comm_spawn 19
2.3.3 MPI_Comm_connect 20
2.4 Designing Benchmarks for Dynamic Process Management 21
2.4.1 Spawn Latency 21
2.4.2 Spawn Rate 22
2.4.3 Inter-communicator point-to-point latency 22
2.5 Distributed Rendering with Dynamic Process Management 23
2.6 Performance Evaluation 23
2.6.1 Spawn Latency 24
2.6.2 Spawn Rate 26
2.6.3 Inter-group Latency 28
2.7 Application-Level Evaluation 29
2.8 Related Work 30
2.9 Summary 31

3. Designing MPI-2 Remote Memory Access Interface 33

3.1 Background 34
3.1.1 MPI-2 Remote Memory Access 34
3.1.2 Overview of InfiniPath 36
3.2 Design of Passive Remote Memory Interface 37
3.2.1 Overview 37
3.2.2 Basic Design (BD-RMA): 38
3.2.3 Helper Thread Design (TH-RMA) 39
3.3 Design of Passive Remote Memory Interface using I/OAT 42
3.3.1 Overview of I/OAT 42
3.3.2 I/OAT Copy Offload Design 42
3.3.3 I/OAT Based Design (IO-RMA) 43
3.4 Performance Evaluation 46
3.4.1 I/O AT Micro-benchmarks 46

3.4.2 MPI Benchmarks 48
3.4.3 Computation-Communication Overlap 50
3.4.4 Effect on Caches 52
3.5 Related Work 53
3.6 Summary 54

4. Conclusions and Future Work 55

4.1 Designing MPI-2 Dynamic Process Management 55
4.2 Designing MPI-2 Remote Memory Access 56

Bibliography 57

LIST OF TABLES

Table **Page**

2.1 POV-Ray Application Execution Times (in seconds) 29

3.1 L2 Cache misses 52

LIST OF FIGURES

Figure		Page
1.1	InfiniBand Architecture	4
2.1	Inter-communicator	12
2.2	Dynamic Process Management framework	15
2.3	Flowchart of Spawn	18
2.4	512 cores: Cyclic rank allocation	25
2.5	512 cores: Block rank allocation	26
2.6	Spawn Rate	27
2.7	Inter-group Latency	28
3.1	Overview of the RMA operations	37
3.2	Basic RMA Design (BD-RMA)	39
3.3	Helper Thread Design (TH-RMA)	40
3.4	I/OAT Copy offload architecture	43
3.5	I/OAT RMA Offloading design (IO-RMA)	44
3.6	Copy Latency (small)	47
3.7	Copy Latency (large)	47

3.8 MPI_Put Latency (small) 48

3.9 MPI_Put Latency (large) 49

3.10 MPI_Put Bandwidth 50

3.11 Computation-Communication Overlap 51

CHAPTER 1

INTRODUCTION

The field of supercomputing has evolved and changed rapidly over the years and the ever increasing network and processor speeds has fueled this growth. The current fastest supercomputer, the Roadrunner [1] has ushered us into a new era achieving a performance of 1026 teraflops, becoming the first machine to break the petaflop barrier. The Roadrunner, like many other clusters on the top 500 [3] list is built from commodity parts. To build a high-performance cluster, designers need to decide on three main aspects: Processor, Interconnect and Programming Model.

The complexity of single processors have reached such a limit that designers are now opting to design multi-core processors to exploit the inherent thread level parallelism in large applications. Scientific applications that have historically run on multiple-processors are well adapted to running on multi-core architectures. The growing number of core counts on the processors have led to dramatic increase in cluster sizes. The Ranger [21] cluster, for example utilizes only 3936 processors but have a total of 62976 processing cores.

The interconnect defines the connection protocols and standards used to connect the processors in a cluster. Some of the common interconnects used are Ethernet, Myrinet, InfiniBand, Quadrics etc. InfiniBand is a high-performance interconnect

based on open standards. InfiniBand has gained widespread acceptance in high performance computing, it is evident by increased usage of InfiniBand in the top 500 [3] list.

A programming model is a software technology used to implement/design parallel algorithms. pthreads, OpenMP, Message Passing Interface (MPI)[12], Partitioned Global Address Space (PGAS) languages are some of the common parallel programming models in use. Of these, MPI has become the de-facto standard for writing parallel programs on large distributed memory machines or clusters. MVAPICH2[15] is a popular MPI implementation over InfiniBand interconnect. It is currently used by over 900 organizations and is used in several of the top 500 supercomputers.

The rest of this chapter provides an overview of modern interconnects, the InfiniBand interconnect and MPI.

1.1 Overview of Modern Interconnects

Modern interconnects are different from traditional interconnects like Ethernet in the features they provide. Without discussing any particular interconnect, we can consider the following features as elements a modern interconnect would possess: Direct Memory Access (DMA) support, support for atomic operations, remote DMA support, connection-less/connection-oriented transport support and reliable/un-reliable transport support. Traditional interconnects had high latency and low bandwidth and high processing overheads. The network protocol itself was run on the host processor and processor performed the costly memory copy operations. Additionally, traditional network devices are controlled by kernel-based drivers and context switches between kernel and applications limited the achievable bandwidth. Modern

interconnects are being designed to support operating system bypass semantics with single-copy schemes. InfiniBand is a modern interconnect that provides all of the above features.

1.2 Overview of InfiniBand

InfiniBand was developed as a system wide network meant to connect CPU's and all I/O devices. InfiniBand was meant to replace all existing fabrics, however it is now used mostly as an inter processor communication fabric in high performance computing. The low latency (1.0-3.0 usec) and high bandwidth ensured rapid adoption of InfiniBand in the cluster and this is evident from the number of clusters in the top 500 (over 25%) that use InfiniBand as its interconnect. InfiniBand adapters can perform all the protocol processing in hardware, such adapters are called Offloaded adapters. Some adapters use the host processor to offload protocol and communication processing; these adapters are called Onloaded adapters. The following sections introduce the communication model and transport modes of InfiniBand.

1.2.1 Communication Model

The InfiniBand network device is referred to as a Host Channel Adapter (HCA). The InfiniBand communication model uses two queues called the send queue and the receive queue. Together, they constitute the queue pair (QP). Send and receive requests are posted in these queues. The work requests (WRs) are posted in the form of a Work Queue Element (WQE) in the send/receive queues. They are picked up by the HCA and acted upon. In the basic send/receive model of communication, the receiver pre-posts several buffers into which the HCA copies incoming data. On completion of any operation, the completion is indicated by placing a completion

entry in the completion queue (CQ). Completion of a send or receive operation can be detected by polling the completion queue or by using an event-based asynchronous notification model. Figure 1.1 shows the InfiniBand communication model.

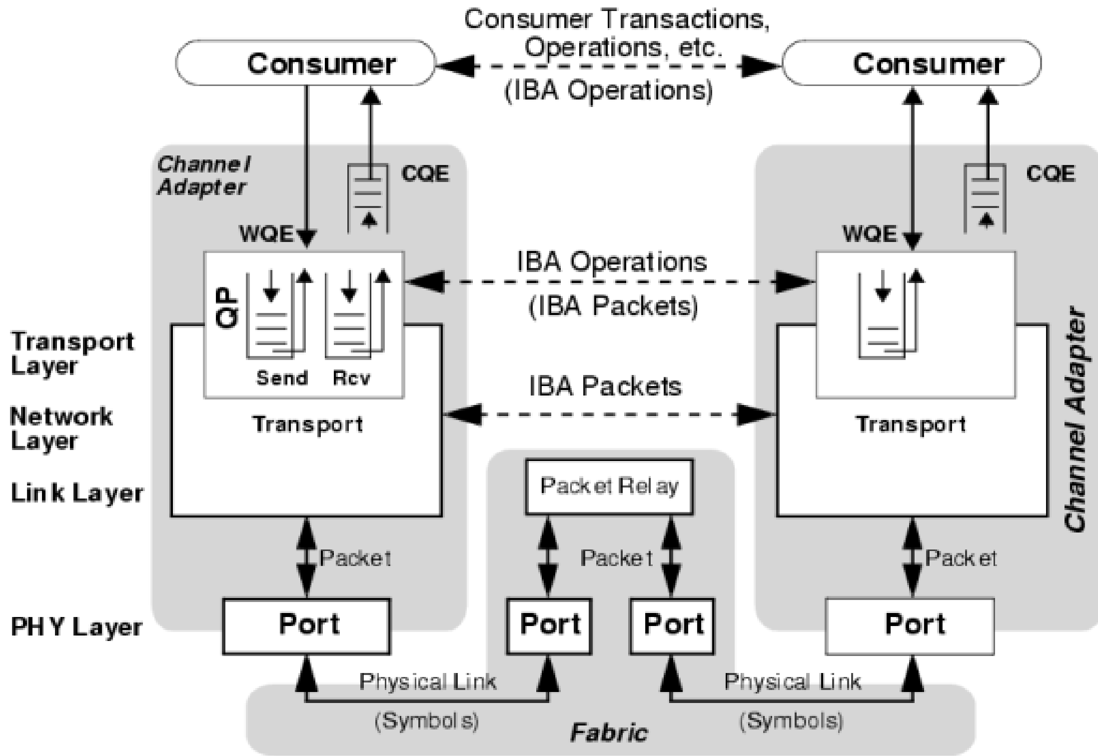

Figure 1.1: InfiniBand Architecture

1.2.2 Transport Modes

The InfiniBand standard defines four modes of transport. Reliable Connection (RC), Unreliable Datagram (UD), Reliable Datagram (RD) and Unreliable Connection (UC). The RC and UD modes are minimally required to be considered an InfiniBand compliant HCA. RD is currently not implemented by any device.

The RC mode is a connected reliable model (akin to TCP) and is the most common mode. An RC queue pair can be used to talk to one another RC queue pair. For n processes to communicate, each process needs to create $n-1$ queue pairs to all its peers.

The UD mode is an unconnected unreliable model (akin to UDP). The main advantage of UD is that a single queue pair can communicate with any other UD queue pair in the network. Since the queue pairs are not connected, each message queued for transmission is also provided the destination parameters. To address a peer, the peers QP number and the Local Identifier (LID) is used. The LID is similar to an IP address. The demerit of UD is that reliability needs to be ensured by the application. Additionally, packetization of data needs to be performed for any data above the Maximum Transfer Unit (MTU) which is typically 2KB.

1.2.3 Offloaded InfiniBand Interfaces

Offloaded InfiniBand HCA's relieve the host processor from performing the network communication processing. Offloaded adapters have an embedded processor on the adapter which performs all the protocol processing. This model allows the application to perform communication and computation concurrently. The InfiniHost HCA's from Mellanox[2] are an example of an offloaded adapter. Offloaded InfiniBand adapters typically have the following features which are lacking on onloaded adapters:

- Remote Direct Memory Access (RDMA): Offloaded adapters can directly write data into memory using DMA transfers. RDMA additionally allows a process direct access to a remote processes memory. The source process sends data as

well as the remote memory address that the data is to be written to. This eliminates the remote processes involvement in the communication operation, thus reducing host CPU overheads. The main disadvantage of the RDMA model is it requires the destination memory to be pinned in physical memory.

- User-mode Operation: Offloaded adapters typically allow user mode applications to initiate sends and receives without trapping into the kernel. Avoiding system calls reduces the number of context switches and improves latency.

- Protocol Offload: Offloaded adapters usually perform all the protocol processing in the adapter itself. This reduces the load on the host processors.

1.2.4 Onloaded InfiniBand Interfaces

Onloaded adapters are similar to traditional Ethernet interfaces. They do not contain any processor to offload protocol operations. The main benefit of this model is that the host processors are generally very powerful and onloaded adapters can leverage this power. Host processor speeds have been growing continously, the arrival of multicore processors now allows the onloaded adapters to perform packet processing as well as computation simultaneously. The InfiniPath[19] adapters from QLogic are based on this model. The InfiniPath adapters do not use DMA to move data from memory to the interface, instead they rely on the programmed I/O memory interface. Since no DMA is performed, the software need not perform any memory pinning. We provide more information about the InfiniPath adapter in Section 3.1.2 when we design of the MPI-2 remote memory access interface for InfiniPath.

1.3 Overview of Message Passing Interface (MPI)

Message passing is a generic form of communication mainly used in parallel computing for inter-process communication. The MPI standard is one such message passing standard defined by the MPI forum[13]. Several implementations of the MPI standard exist. MPICH, MPICH2, MVAPICH/MVAPICH2, OpenMPI and HP-MPI are some of the major MPI implementations.

The first version of the MPI standard was called the MPI-1. The MPI-1 standard defined the basic MPI point-to-point interface, the MPI collective interface and other glue interfaces. The second version of the MPI standard, called MPI-2 was defined in 2003. MPI-2 introduced several extensions to the existing MPI-1 specification. It defined the MPI-2 dynamic process management interface, the MPI-2 remote memory access interface and the MPI File I/O interfaces.

We first provide a basic overview of the MPI communication model, point-to-point and collective interfaces. An overview of the remote memory access interface and the dynamic process management interface are presented in Section 2.1 and Section 3.1 respectively.

1.3.1 MPI Communicators

In an MPI program, two processes can exchange messages. An MPI process is identified by a [rank, process group] pair. The MPI communicator encapsulates the ranks and the process group for which the ranks are described. Thus, a communicator is a software construct that defines a group of processes and a context identifier for communication within that group. All MPI operations are performed in the context of some communicator. MPI operations use the rank and communicator context

information to deliver messages to the target. The MPI_COMM_WORLD is a predefined communicator that allows for communication between all processes of the job.

1.3.2 Point-to-point communication

Two MPI processes can exchange messages using the point-to-point communication interface. The sender process sends the message using the function MPI_Send. The MPI_Send API requires the sending buffer, size of data being transmitted, the type of the data (INT, FLOAT, DOUBLE etc), a unique send tag, the receiver's rank and the communicator to be used. The receiver uses the MPI_Recv function to receive the data into its buffer. It provides the length of data to receive, the type of the data, the unique tag, the sender's rank and the communicator. The message is matched at the receiver using the [tag, source, communicator_context] three-tuple. Both MPI_Send and MPI_Recv are blocking functions, meaning they don't return until the data has been sent or received respectively. Alternately, MPI provides the MPI_Isend and MPI_Irecv interfaces to perform non-blocking send/receive. These function initiate the send or receive and return immediatly. The send/receive progresses asynchronously. The application is required to perform a MPI_Wait to complete the operations. If the application does not perform the MPI_Wait the send/receive might not progress at all (Note: this is implementation dependent and varies across libraries).

1.3.3 Collective communication

A collective operation are communication operations done by a group of processes acting together. MPI offers several types of collective operations. For example: MPI_Bcast is used to broadcast a certain message from one rank to all others, MPI_Alltoall is used by a group of processes to send data to all their peers and receive some data from the peers. MPI_Barrier, MPI_Reduce, MPI_Allreduce, MPI_Allgather are some of the other collectives.

1.4 Problem Statement

The MPI-2 standard introduced two new programming interfaces. The *dynamic process management* interface and the *remote memory access interface*.

With the emergence of large distributed computing interfaces like MapReduce and Hadoop, the grid paradigm has become important. The MPI-2 dynamic process interface can be used in distributed computing or grids for performing computation. However, the MPI-2 dynamic process interface has not been well studied in current literature. With regard to this, we attempt to address the following issues:

- How can we design an efficient dynamic process interface on InfiniBand?
- Can we run real-world applications using dynamic process interface without significant performance losses ?

The other important MPI-2 feature is the remote memory access interface. With the increased popularity of new PGAS programming models like UPC and X10 it is important to address the issue of designing MPI-2 remote memory access interface on modern interconnects. In particular we attempt to address the issues of designing

MPI-2 remote memory access on onloaded adapters. As mentioned earlier onloaded adapters lack the RDMA feature, without RDMA, the remote memory access interface needs to be designed with point-to-point primitives. We attempt to address this issue and propose new designs for better remote memory access over point-to-point primitives. We address this in two steps:

- What are issues with the current MPI-2 remote memory access designs ?
- How can we design efficient MPI-2 remote memory access interface on Onloaded InfiniBand adapters?

1.5 Organization of Thesis

The rest of this thesis is organized in the following way. Chapter 2 we design the MPI-2 dynamic process interface over InfiniBand. We first present the dynamic process model and the requirements of the design. We propose multiple designs and evaluate our designs using benchmarks. Due to the lack of benchmarks in this area, we have also designed these benchmarks. Finally, we show the performance of a real-world ray-tracing application which uses the dynamic process interface.

In Chapter 3 we present the basic designs for MPI-2 remote memory access. We study the issues with the naive designs and present new designs. Finally, we evaluate our designs using micro-benchmarks. Conclusions and future work are presented in Chapter 4.

CHAPTER 2

DESIGNING MPI-2 DYNAMIC PROCESS MANAGEMENT

Dynamic process management is a feature of MPI-2 that allows an MPI process to create new processes and manage communication with these processes. The dynamic creation of processes allows application writers to develop multi-scale applications or master/worker based programs. Although several MPI implementations support this feature there have not been any studies in designing the dynamic process interface. Also, there are no standard benchmarks or applications that can evaluate the dynamic process interface.

In this chapter, we present a design for the dynamic interface on InfiniBand. The rest of this chapter is organized as follows: Section 2.1 presents the background information- an introduction to the dynamic interface. Section 2.2 presents our proposed design followed by Section 2.4 which proposes a set of benchmarks which is used to evaluate our designs. In Section 2.6 we evaluate our design and present experimental results and finally conclude and summarize this section in Section 2.9

2.1 Background

2.1.1 MPI Inter-communicator

In Section 1.3.1 we introduced the concept of an MPI communicator. The communicator described is what typically called an intra-communicator: intra because it allows for communication between processes in a certain process group. MPI-2 provides another type of communicator called the inter-communicator. The inter-communicator is the basis of operation of the dynamic process model. An inter-communicator is so called because it allows for communication between two process groups. All communication is performed from a process in the local group with a process in the remote group. Figure 2.1 illustrates the working of an inter-communicator. The data is being sent from rank 0 from the left process group (called local group) to the rank 0 on the right process group (called the remote group).

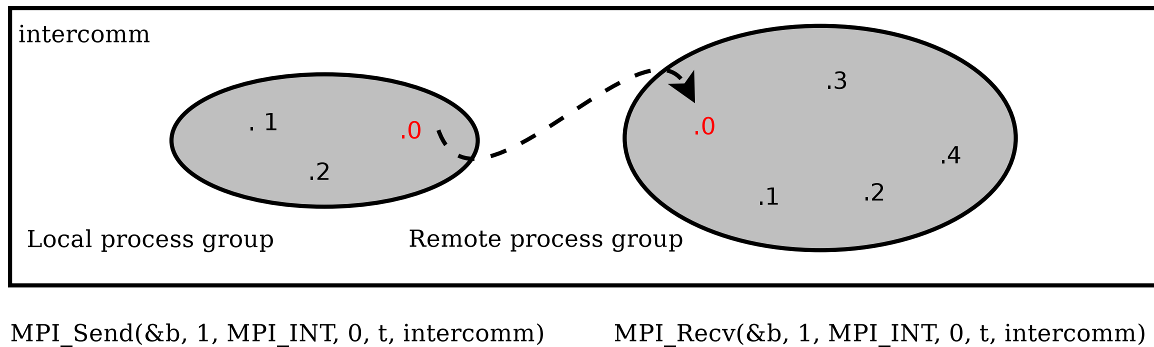

Figure 2.1: Inter-communicator

The MPI-2 dynamic interface uses intra-communicators to connect two existing intra-communicator process groups into a single inter-communicator, thus creating communication between process groups. This allows an existing MPI job to spawn a

new set of processes: the remote group. The two groups are bound with the inter-communicator and can now exchange MPI messages. Furthermore, MPI allows us to create a new intra-communicator from the inter-communicator thus merging the process groups into a single large group. (thus growing the processes in the MPI job).

2.1.2 Dynamic Process API

The MPI standard defines three ways of creating or joining new processes into existing MPI jobs. In our description of the interface we call the spawning process parent process-root and the root of the spawned group the child-root.

- **MPI_Comm_spawn:**

  ```
  int MPI_Comm_spawn(char *command, char *argv[], int maxprocs,
  MPI_Info *info, int root, MPI_Comm comm, MPI_Comm *intercomm,
  int array_of_errcodes[])
  ```

  The function starts *maxprocs* copies of *command* process. The function is collective over the communicator *comm*, i.e. the function does not complete until all processes in the communicator have created the inter-communicator and the *root* process performs the role of the parent-root. The newly created set of processes form an MPI_COMM_WORLD of their own. The root of the spawned process group uses the function *MPI_Comm_get_parent* to discover if it was spawned from an existing MPI process. The API uses the accept/connect interface between the parent-root and child-root to exchange rank and process group information. The function returns *intercomm*, an inter-communicator which contains the spawns in the remote group. The MPI application can now exchange messages with the newly created processes using intercomm. The MPI standard does not specify where and how the processes were started and leaves

it to the job scheduling infrastructure to manage. It only provides a information structure *info* to propagate any hints to the job scheduling framework.

- **MPI_Comm_accept/MPI_Comm_connect:**

  ```
  int MPI_Comm_accept(char *port_name, MPI_Info info, int root,
  MPI_Comm comm, MPI_Comm *newcomm)
  int MPI_Comm_connect(char *port_name, MPI_Info info, int root,
  MPI_Comm comm, MPI_Comm *newcomm)
  ```

  This API facilitates a client-server computing model to MPI processes with the server process using the *MP_Comm_accept* to wait for incoming connection on *port_name*. The client uses *MPI_Comm_connect* to connect to the *port_name*. *port_name* is an implementation and interconnect specific string that identifies a process. The resulting inter-communicator *intercomm* now allows the client process group to exchange messages with the server process group. As with MPI Comm spawn, the accept/connect calls are collective over the communicator *comm* and the *root* act as the root ranks in the connection establishment. Once the connection is created, both process groups can communicate with the remote groups using the inter-communicator.

- **MPI_Comm_join:**

  ```
  int MPI_Comm_join(int sockfd, MPI_Comm *intercomm)
  ```

  Using this interface, two processes with an existing TCP/IP connection described by the socket *sockfd* can establish an inter-communicator and start MPI message exchange. The inter-communicator describes a singleton local group and a remote group in this case. The socket is used to exchange MPI port information, followed by an MPI connection creation using the accept/connect interface. The socket is never used for MPI communication.

2.2 Design of Dynamic Process Management

In this section we describe our design of the dynamic process management framework. We first consider the architecture of the dynamic process framework.

2.2.1 Dynamic process framework

Figure 2.2 hows the architecture of the MPI-2 dynamic process interface.

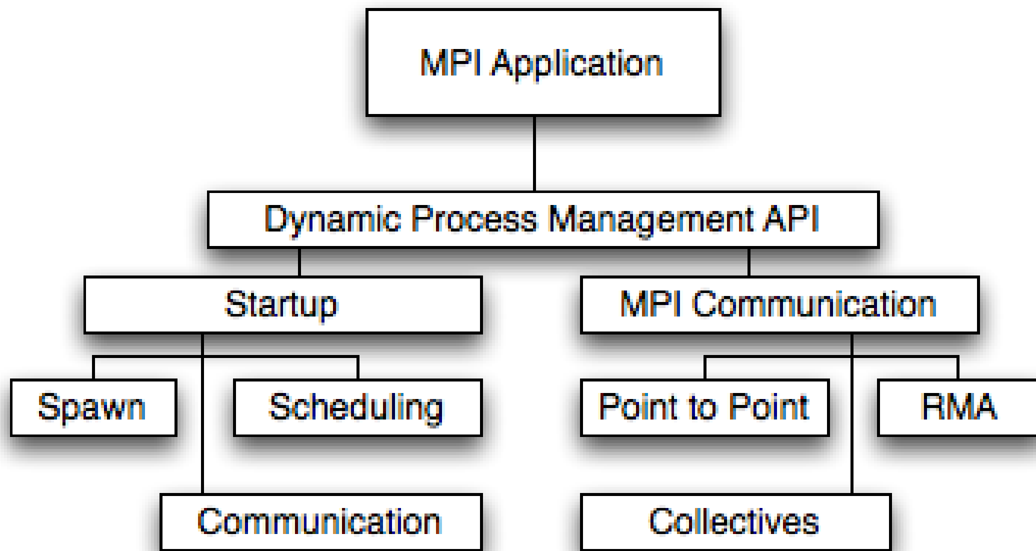

Figure 2.2: Dynamic Process Management framework

The MPI application uses the API described in Section 2.1.2 to spawn new tasks. An MPI design has to handle the startup of the new tasks and the three phases of this startup are the spawn phase, scheduling phase and the communication phase.

2.2.2 Spawn Phase

The spawn design requires the MPI application talk to the job manager. This is accomplished using a common protocol between the dynamic process management API and the job launcher. We use the Process Management Interface (PMI), a generic protocol used by MPICH2[14] and MVAPICH2 [15] to communicate with the job launcher. In our designs, we consider two job launch schemes, the Multi-Purpose Daemon (MPD), which is the default scheme in MPICH2 [10] and mpirun_rsh, a MVAPICH2 specic startup manager based on the ScELA [20] architecture. The PMI protocol uses key-value based message broadcast primitives to propagate information. The parent-root broadcasts its port information, size of the job, the actual binary to launch and arguments to the job-launcher via the PMI primitives. The job-launcher takes corresponding action by performing the actual launch of the binaries on the available hosts. The job-launcher has to perform a scheduling decision, and this is the second phase of the startup.

2.2.3 Scheduling Phase

MPI-2 standard does not define a way to do task placement. The task scheduling is performed by the startup agent or a job management system. Scheduling of dynamic tasks requires the job manager to maintain global history of dynamic tasks and place tasks based on this history. Our implementation uses MPD or mpirun_rsh to schedule tasks. Both tools place tasks in a round-robin manner, but suffer from the drawback that multiple spawns are scheduled in the same order of available nodes resulting in imbalance. The studies in [4] have addressed this issue in LAM-MPI by suggesting various task placement mechanisms to maintain load balance. The third and last

phase of the dynamic interface is the communication phase. We provide a detailed design for this phase.

2.3 Design of Dynamic Process Management: Communication Phase

To design the spawn interface we require the parent to request a spawn and wait in the MPI_Comm_accept interface to establish the inter-communicator. The communication phase begins with the child-root of the spawned process group connecting back to the parent to exchange process group information. To establish the inter-communicator the processes need to know the process group ID, the size of the remote process group and the context ID to be used. Additionally, implementations may require a way to identify each remote rank independently to exchange messages. In our design each rank is uniquely identified by their UD queue pair numbers and the LID. This information is exchanged between the root processes and broadcast within their local groups. Figure 2.3 shows the flow of information required to implement the spawn interface.

2.3.1 Communication Methods

Every spawn requests results in the child-root connecting to the parent process to exchange information. An application that spawns tasks frequently will incur the overhead of this connection establishment and communication for every spawn. Thus, to efficiently design the spawn interface we need lightweight connection establishment protocols and as noted in Section 1.2.2 there are different transport modes for Inni-Band that we can use for this designing this phase:

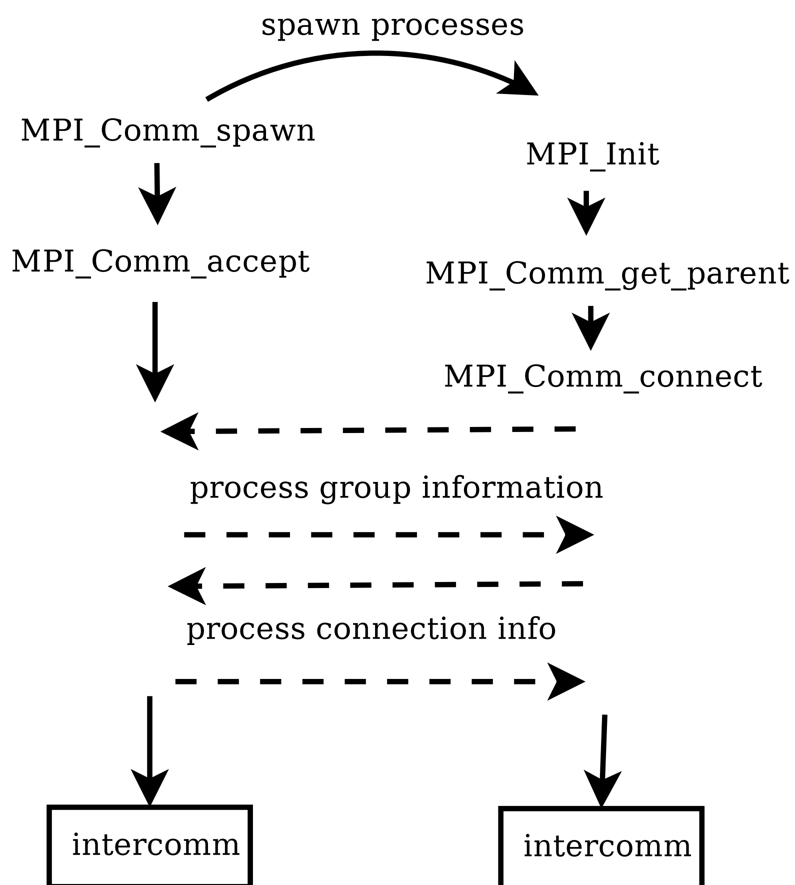

Figure 2.3: Flowchart of Spawn

- Reliable Connection (RC):

This is a reliable but connection-oriented mode. There is significant overhead to communicate with a new process. Establishing a reliable connection requires the process to trap to the kernel and request a new queue pair. However, this is a one-time cost and once this is performed using RC can provide lower latencies due to other features such as RDMA.

- Unreliable Datagram (UD):

Unreliable Datagram (UD): Unreliable and connectionless. There is a very low overhead to communicate with a new process. Software must perform segmentation of messages over the MTU (often 2KB). If the amount of data to be exchanged between the local and remote roots is small then using UD provides benefits. Since the data size is small providing segmentation is cheap and there is no connection overhead. As the number of spawned process in a group goes up, the data size will increase. In this case using the RC may provide a benet.

2.3.2 MPI_Comm_spawn

To perform the spawn, we first create the connection information of the parent that is passed to environment of the spawned children. This is managed via environment variables and propagated by the job manager. The parent process advertises a port in the form of an LID and two UD queue pair numbers. One of the UD queue pair numbers is utilized for the accept/connect interface. The other UD queue pair number is used for RC QP connection establishment [25]. Once the processes are spawned, the parent process waits for the child-root of the remote group to connect back.

2.3.3 MPI_Comm_connect

The spawned process group collectively performs the connect. Only the child-root connects to the parent process, while the other ranks wait for remote group information. We have two possible designs at this point, using RC for message exchange versus using UD.

- UD

If the amount of data to be exchanged with remote root is small then it is more efficient to use a direct UD exchange. In this mode, the child-root sends the process group size, process group ID and context ID for the communicator in a single UD message. The parent-root acknowledges the exchange and sends its process group ID, group size and context ID. Both the ranks broadcast the remote group information within their own MPI COMM WORLD. In the next step, both root processes exchange the connection information within their local groups. In our design the connection information consists of the LID and UD QPN. In applications that spawn often and spawn few processes the UD direct exchange model is more scalable and quicker than creating short-lived RC connections.

- RC

If an application spawns large jobs and spawns are infrequent, the connect API uses the second UD QP number to establish an RC connection with the remote root. This connection establishment is according to the algorithm defined in [25]. Following the message exchange, the two root ranks establish a RC connection that is used to exchange process group information. At the end of the above stage each process

has the information required to independently create the inter-communicator to communicate with the remote group. The inter-communicator can now use regular MPI communication using the point-to-point, remote memory access (RMA) or collectives.

We described the design of the dynamic process interface. To evaluate the performance of our design we need to design a set of benchmarks. To the best of our knowledge, there are no benchmarks to evaluate the MPI-2 dynamic process interface. The following section describes our benchmarks.

2.4 Designing Benchmarks for Dynamic Process Management

To the best of our knowledge, there are currently no metrics or standard applications to benchmark various designs and implementations of MPI-2 dynamic process management. To address this need we design a set of benchmarks that are useful to measure performance of a MPI-2 library. The benchmarks are similar to the existing OSU Benchmark suite [17] released with the MVAPICH/MVAPICH2 software.

2.4.1 Spawn Latency

The spawn latency benchmark measures the time taken to perform the MPI Comm spawn routine. We time the execution of this function in the parent-root process. The time to spawn is an important metric as it is the measure of the overhead in using dynamic process management. Minimizing this overhead is vital if dynamic processes are to be used in MPI applications. Due to involvement of system resources and job manager framework, the measured values of the latency has signicant variation. The benchmark averages the latency over a large number of runs.

2.4.2 Spawn Rate

The spawn rate benchmark measures the rate at which an implementation is able to perform the MPI Comm spawn routine. It is calculated by spawning jobs continuously and finding the rate at which the implementation is able to create new MPI jobs. The benchmark does not consider the time for disconnecting of the inter-communicator. Spawn Rate is an important metric as it can estimate the scalability of our design. To minimize the effect of spawned jobs on the spawn rate we put the spawned process to sleep until the benchmark is complete. This is required as multiple jobs will be scheduled to the same cores as the benchmark progresses.

2.4.3 Inter-communicator point-to-point latency

Sending point-to-point MPI data across an inter-communicator requires us to send data from a local group to a remote group. This inter-group message latency is an important metric as designs may have better optimizations for intra-communicators than inter-communicators. With inter-communicators, message delivery has an additional overhead of mapping from the (local process group, rank) to the (remote process group, rank). In some designs, such as ours, no connections are setup between ranks of the local and remote process groups. Connections are setup on-demand, when the ranks really need to communicate. The benchmark thus measures the effective latency due to the connection establishment and the data transfer. The inter-group latency calculates an average latency for a range of data sizes, between two ranks.

2.5 Distributed Rendering with Dynamic Process Management

Graphics rendering is a highly parallelizable activity. Distributed rendering works by distributing each frame to be rendered to the compute nodes of a cluster. A frame can usually be rendered independently of other frames and the only communications involved are the initial frame data distribution and final collection of rendered images. Rendering can programmed easily using a master-slave model. Render farms are common in Computer Graphics Imagery (CGI) industry, with the farms hosting several render servers that can be used by clients. To demonstrate the feasibility and real-world application of the dynamic process interface we designed a dynamic process version of POV-Ray, a popular, open-source ray-tracing application. Using our design, a graphics programmer can decide at execution time the optimal number of compute nodes required for the job and spawn the rendering on the nodes. There have been MPI parallelization efforts on POVRay [18], but these implementations use a static runtime environment. The dynamic process interface can be programmed to have an changing environment in which we can expand or contract the available slave resources. This is similar in concept to a render farm and this paradigm can be programmed using the MPI-2 interface. We will present our evaluations with POV-Ray in Section 2.7.

2.6 Performance Evaluation

We use a 64-node Xeon cluster with each node having 8 cores and 6 GB RAM. The nodes are equipped with InniBand DDR HCAs and 1 GigE NICs. We present results using a 64x8 layout, which uses all 512 cores, with cyclic allocation of ranks. We

also present a result with block allocation of ranks. Our designs were implemented in the MVAPICH2 library. We evaluate our design in MVAPICH2 as well as OpenMPI, another popular MPI library.

2.6.1 Spawn Latency

Figure 2.4 shows the results of running the spawn latency benchmark. We present five results in the graph, mvapich2MPD-RC: which uses only RC connections and MPD for startup, mvapich2-mpirun rsh-RC: which uses RC connections and mpirun rsh for startup, mvapich2-MPD-UD, which uses UD for initial information exchange and MPD for 5 startup and mvapich2-mpirun rsh-UD which uses UD for initial information exchange, mpirun rsh for startup and OpenMPI : which shows the latency results for the OpenMPI library.

As seen in Figure 2.4 the RC and UD implementations perform almost equally when MPD is used for very small job sizes. For job size of 32 and beyond the UD design shows a slight benet. With mpirun_rsh we see a that the UD design provides a lower spawn latency. The mvapich2-mpirun_rsh-RC and OpenMPI perform similarly (up to 128 processors) as both use a connection based startup model with similar job launch mechanism. However, for 512 processes, mvapich2-mpirun_rsh designs perform better than OpenMPI. On the job startup angle, we find the MPD startup mechanism is faster than mpirun_rsh for small job size, however for larger jobs mpirun rsh is more scalable. This is due to the fact that MPD maintains a ring-of-daemons on all nodes. Spawning a new job on a node just requires a TCP/IP message to be sent to the daemon, as compared to mpirun_rsh which requires us to first spawn the launcher on each node followed by the launcher spawning the new processes. MPD, however has

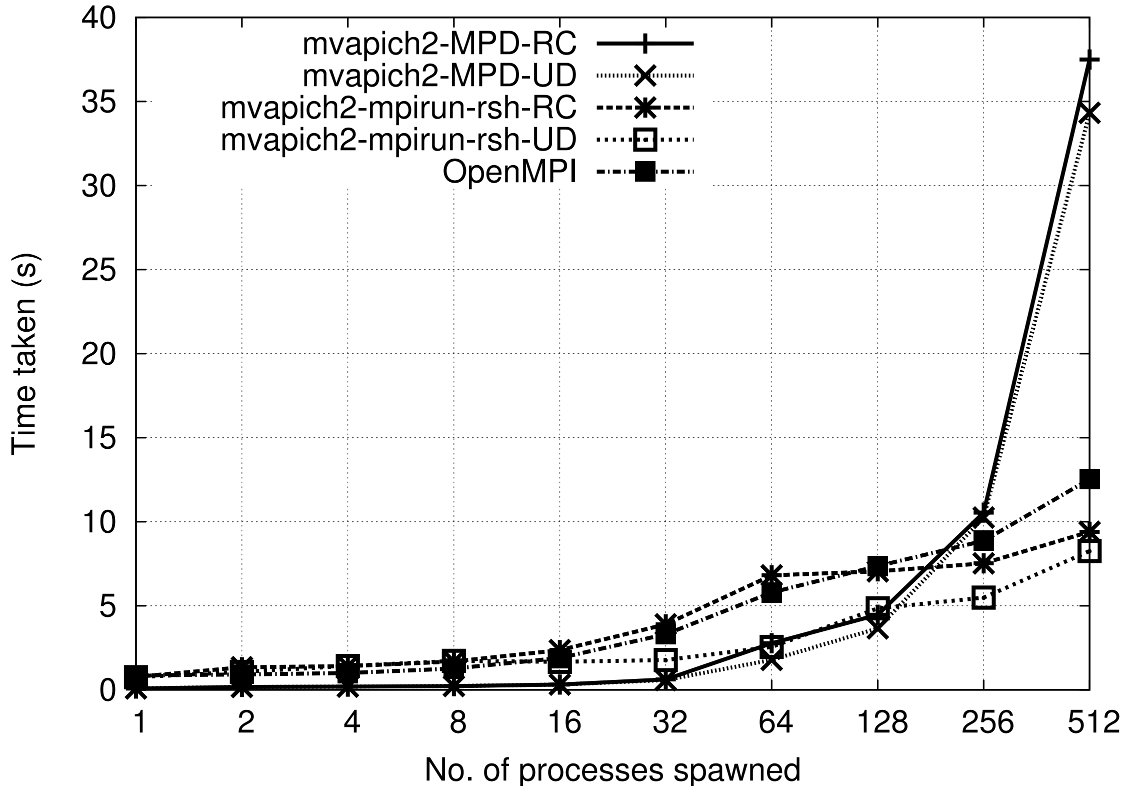

Figure 2.4: 512 cores: Cyclic rank allocation

'

higher startup latency as number of ranks grows. mpirun_rsh is a daemon-less startup manager based on ScELA architecture [20]. It incurs higher overhead for small job launches, but it is highly scalable and provides very low latency for higher job sizes.

The second set of results we present in Figure 2.5 are the the spawn latency with block allocation of ranks. This is an important result as it shows the effect of HCA contention on the spawn time. As seen in the graph, when there are multiple jobs per node, the UD spawn design performs better than the RC design, as the UD model has lesser startup overhead. The UD design is more relevant here as job allocation is generally block distributed. The UD design is simpler and lightweight. OpenMPI

performs very similar to mvapich2-mpirun_rsh-RC in this benchmark for up to 256 processes. However, for 512 processes mvapich2-mpirun_rsh designs perform the best.

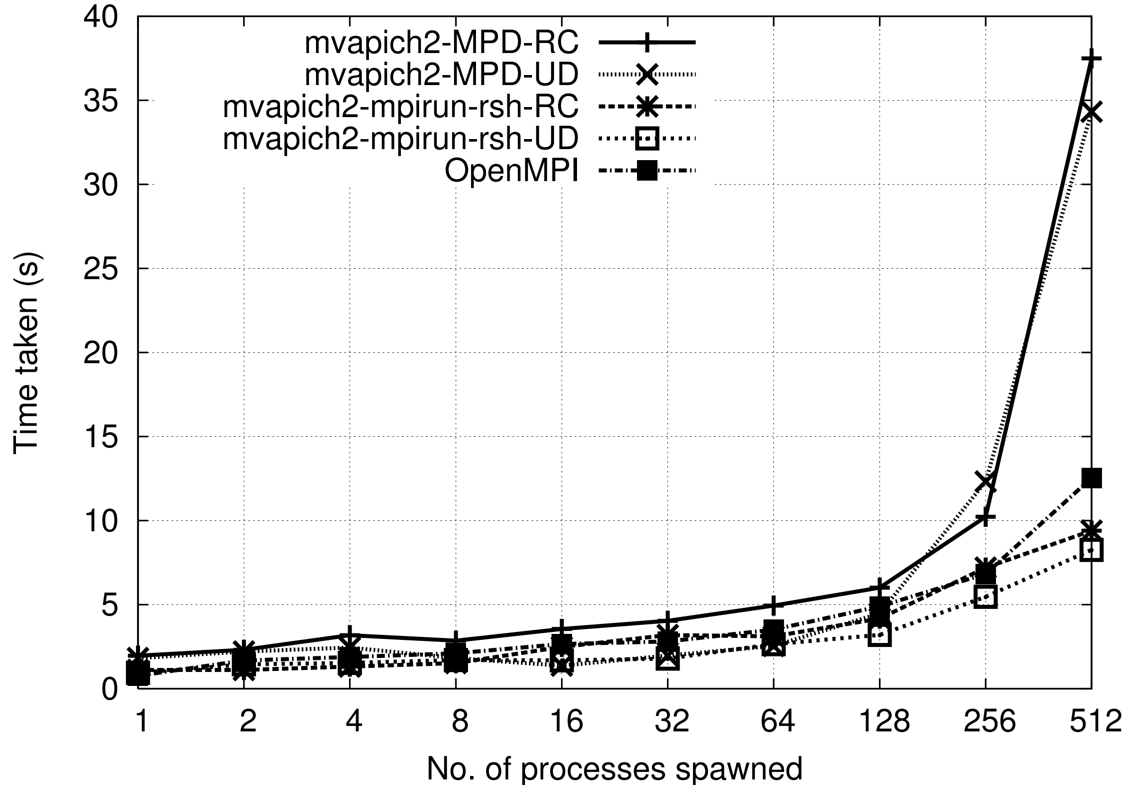

Figure 2.5: 512 cores: Block rank allocation

2.6.2 Spawn Rate

The spawn rate benchmark is evaluated with 16-nodes of the cluster, for a total of 128 cores. The benchmark measures the rate of sustained spawn supported by our design. The reported value is the number of spawns/second with increasing job sizes. Figure 2.6 shows the results of the benchmark running on our design.

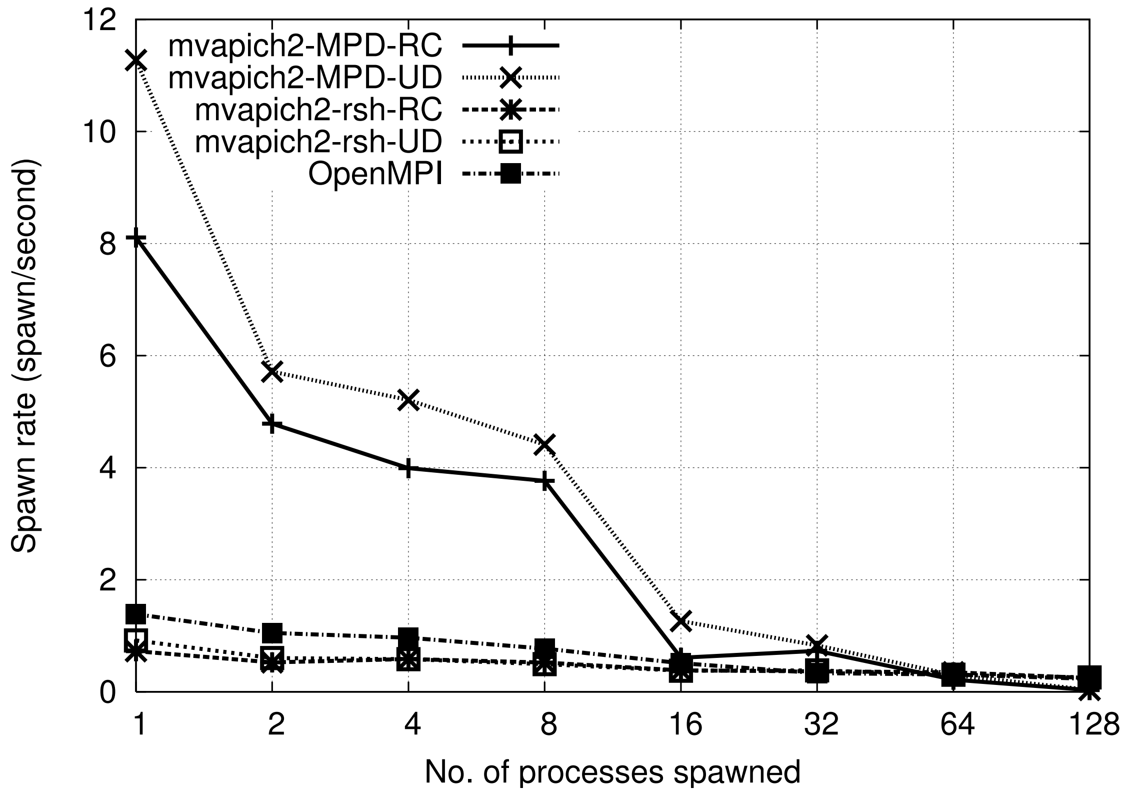

Figure 2.6: Spawn Rate

We see that the UD design using MPD job manager provides the best spawn rate. The relatively higher cost of creating and destroying RC queue pairs leads to a slower spawn rate with RC. As we have seen mpirun rsh startup has a higher initial overhead and results in a lower spawn rate, however it scales very well and maintains a steady spawn rate with increasing job size. OpenMPI performs similar to mpirun rsh and has a low spawn rate for small jobs. Only mvapich2-MPD designs are able to provide a high spawn rate for small jobs. The spawn rate is an important metric to consider when designing an MPI application with frequent job spawns. The benchmark clearly

shows that to have a high spawn rate we need a low-overhead connection mode (like UD) and an MPD-like startup framework.

2.6.3 Inter-group Latency

The inter-group latency is a basic latency test to measure the difference between intra-communicator latency and intercommunicator latency. As we see in Figure 2.7,

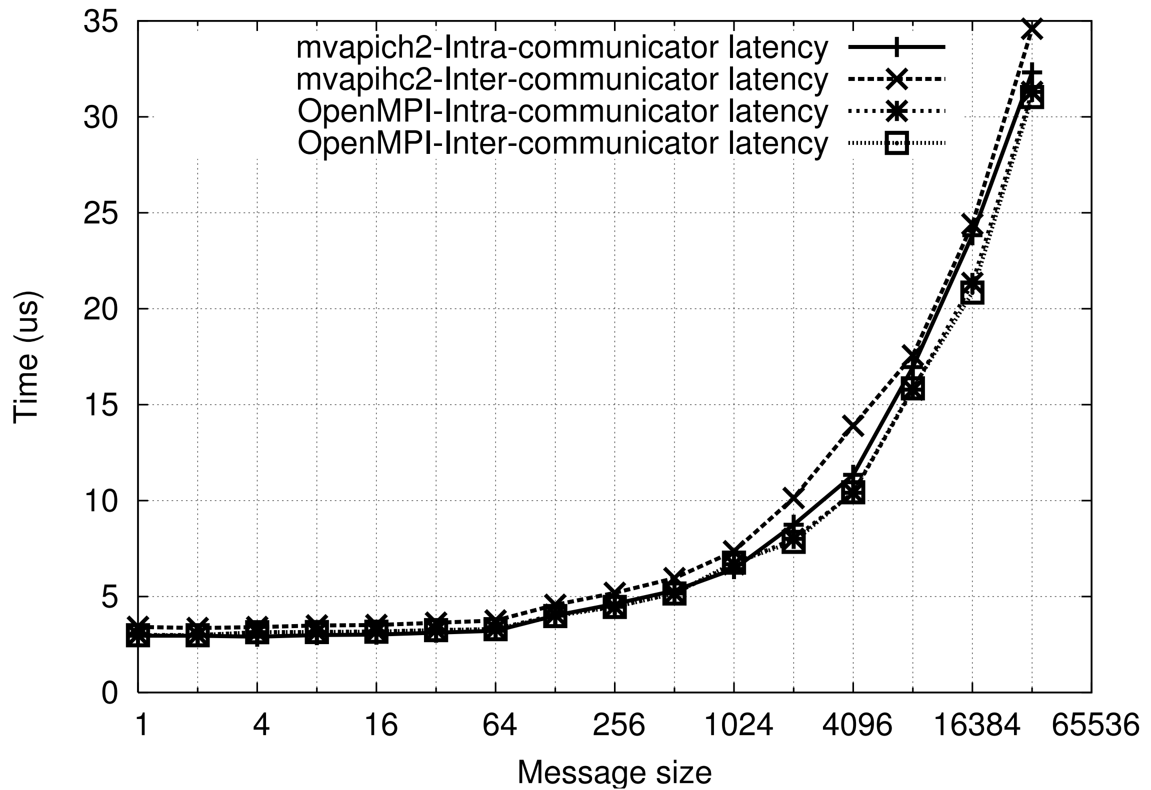

Figure 2.7: Inter-group Latency

for small message sizes, the mvapich2 inter-communicator exchange has a slightly higher latency. This higher latency is due to searching of process group and managing the translation from local group to remote group. For large messages, the latency

of both the message exchanges are almost equal with very little variation and the data transfer component dominates and the process group translation cost does not affect overall latency. OpenMPI does not show any difference between intercommunicator and intra-communicators. However, OpenMPI does perform slightly better than MVAPICH2 for larger messages. This is due to the higher rendezvous threshold utilized by the OpenMPI library compared to MVAPICH2

2.7 Application-Level Evaluation

The final results we present are the evaluations of a dynamic process POV-Ray derived ray-tracing application. We implemented a parallel version of POV-Ray to use MPI-2 dynamic process interface. We compare the results of using our RC design, UD design and traditional static runtime parallel POV-Ray. For our evaluation we render a 3000x3000 *glass chess board* with global illumination. Table 2.1 shows the results of our evaluation.

Table 2.1: POV-Ray Application Execution Times (in seconds)

NP	*MPD-RC*	mpirun-rsh_RC	*MPD-UD*	mpirun-rsh_UD	Traditional
2	*2500*	2500	*2494*	2494	2523
4	*1258*	1257	*1251*	1255	1263
8	*631*	635	*634*	639	644
16	*363*	368	*360*	365	342
32	*220*	230	*215*	225	196
64	*148*	160	*144*	154	129

Note: NP means Number of Processors

As seen in the table, the dynamic process framework adds very little overhead to the overall execution of the application. Until 32 processes the speedup factor is

almost the same for all three designs. Beyond 32 processors, the cost of startup and parallelization starts to accumulate and the dynamic version incurs some slowdown.

Evaluating a real-world problem clearly shows the feasibility of the dynamic process framework. Moreover, using dynamic processes give more control to the application programmer who can intelligently decide the parallelization factor and placement of jobs at run-time. Additionally, using the dynamic process framework, applications can dynamically change size and scale of the application which is a key benefit.

2.8 Related Work

The dynamic process architecture was defined by Gropp and Lusk [8]. The MPI-2 standard defined the process creation and management interface. The standard left the scheduling decisions to the MPI implementation. Marcia Cera et al [4] have explored the issue of improving scheduling of dynamic tasks. Their solution perform load-balancing of jobs across nodes of a cluster.

Edgar Gabriel et al. [6] provided an evaluation of the performance of dynamic process interface of popular MPI implementations. However, they do not provide any designs or discuss the issues in designing the interface. To the best of our knowledge, our work is the first one that considers all the issues in designing and provides a detailed design of a high-performance dynamic process interface. Our work is targeted on InfiniBand but the same concepts hold for traditional networks like Ethernet.

Several researchers have explored using the dynamic process interface for fault-tolerance in MPI applications. [11]. Kim et al. [9] have explored the design and implementation of dynamic process management framework for grid-enabled MPICH.

However, their work did not explore the design of the MPI-2 dynamic process interface, but implemented a new MPI interface *MPI_Rejoin* that allows processes to join existing process groups.

2.9 Summary

Over the years, MPI has become the dominant parallel programming model. Traditionally, grid and parallel applications have used the master/slave model of computation. The MPI-2 dynamic process interface can be used in master/slave designs and thus is finding increased adoption in grid environments. Additionally, the connect/accept interface allows MPI-2 to be used in client/server models. In this work we have addressed the designing of an efficient dynamic process interface. We implemented our designs and evaluated them on MVAPICH2 [15], popular MPI implementation on InfiniBand. The lack of benchmarks in this area was addressed and we designed benchmarks to evaluate our designs. Our study draws the following conclusions:

- An MPD-like daemon based startup model is required for supporting frequent task spawning. The spawn rate benchmark clearly shows the superiority of the daemon based startup model.

- MPD suffers from very high latency for large job sizes. For very large job launches, the ScELA [20] architecture has proved to be highly scalable and reliable. Thus, mpirun rsh based startup models are required for managing large jobs.

- Lightweight communication primitives are better for the task startup phase. The benchmarks show the advantage of using a UD model for InniBand. Similar lightweight transport schemes (such as UDP) should apply in other environments (such as 10GigE).

- MPI Applications dont incur heavy overhead in using the dynamic process framework. The evaluation of the ray-tracing application clearly demonstrates the feasibility of the dynamic process paradigm with the benets of dynamically growing or shrinking jobs.

CHAPTER 3

DESIGNING MPI-2 REMOTE MEMORY ACCESS INTERFACE

The MPI-2 Remote Memory Access (RMA) interface provides an alternate programming model, different from the MPI-1 point-to-point operations. In this model, the communicating process knows the target process's memory addresses and can directly perform operations on the target's memory. One of the key advantages of using the remote memory operations is that, a single synchronization is enough to initiate the data transfer of RMA several operations, unlike point-to-point which requires synchronization for every operation. Additionally, due to the de-coupling of communication and synchronization RMA allows processes to achieve computation-communication overlap.

The designs of the MPI-2 remote memory access has been investigated by several researchers. However, most researchers have explored efficient design of the MPI-2 RMA interface on interconnects with Remote Memory Direct Access (RDMA) primitives. Several adapters such as On-loaded InfiniBand adapters and 1/10 GigE Ethernet still lack support for remote operations. On such adapters the design of the RMA interface require active involvement of the target processors thus limiting the performance of the application. In this work we aim to address this area and improve

the design of the MPI-2 remote memory access to provide better performance and overlap. We provide a novel approach using the Intel I/O Acceleration engine to provide computation-copy overlap. We evaluate our designs and show the benefits in terms of improved cache utilization, better overlap and lower latencies.

The rest of this chapter is organized as follows. In Section 3.1 we provide a background of the MPI-2 remote memory interface and the InfiniPath adapter (an on-loaded adapter) on which we perform our evaluations and experiments. Section 3.2 presents the design choices for the MPI-2 RMA interface for on-loaded interconnect. Section 3.3 presents our optimized design with hardware copy-offloading. In Section 3.4 we provide an evaluation of our designs and finally, Section 3.6 presents a conclusion to this chapter.

3.1 Background

3.1.1 MPI-2 Remote Memory Access

In MPI-2 remote memory access (also referred to as RMA or one-sided communication) the origin process (the process that issues the RMA operation) can access a target processes memory address space directly. The origin process. The origin process provides all the parameters such as target rank, target memory address, target datatype, etc. The memory area operated on is known as the *window* in MPI parlance. MPI defines three one-sided operations: *MPI_Put*, *MPI_Get* and *MPI_Accumulate*. To illustrate the usage of the Put, we consider the definition of the Put operation.

```
MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp,
int target_count, MPI_Datatype target_datatype, MPI_Win win)
```

The operation writes the data present in *origin_addr* and of size *origin_count* to the destination. The actual size to be 'Put' is the size of the datatype times the count. The data is written to the window on *target_rank* at a displacement of *target_disp*. The address of the window's are collectively exchanged initially during the window creation phase. The data being written need not be homogeneous, i.e. the target can treat it as a different datatype with a different count, and hence these are also specified by the origin process.

The initiation and synchronization of MPI-2 RMA operations need to be done using MPI-2 synchronization primitives. MPI semantics allows one-sided operations only within an *epoch*, which is the period between two synchronization events. MPI-2 defines two forms of synchronization.

- **Active Synchronization:** this requires both origin and target to synchronize via a collective operation. The *MPI_Win_fence* and *MPI_Win_post/MPI_Win_start* are the active synchronization methods.

- **Passive Synchronization:** In this method the origin process can lock a target window, perform one-sided operations and unlock the window. In the passive synchronization mode, the origin process alone makes the synchronization and issues data transfer operations. The remote process is not involved at all. The *MPI_Win_lock/MPI_Win_unlock* interface is used for this mode.

Passive synchronization is very effective on large scale systems as it can minimize the coordination between the origin and target process. This kind of synchronization gives greater potential for computation/communication overlap. In this work, we primarily concentrate on the passive mode of synchronization. We mainly focus on

designing MPI one-sided communication for net- works that do not support one-sided RDMA semantics. As a case study for this work, we use the InfiniPath [5] network that doesnt expose an RDMA (Remote Direct Memory Access) interface, unlike traditional verbs that support both send/receive as well as RDMA semantics.

3.1.2 Overview of InfiniPath

The InfiniPath Host Channel Adapter (HCA), an InfiniBand adapter from QLogic has two verbs layer over which MPI can be written: the traditional Open Fabrics layer and a light-weight communication layer (called the PSM API layer). InniPath uses a connectionless model for communication. InInfiniPath HCAs do not use DMA engines to send/receive data and thus do not require memory pinning. The PSM API provides communication semantics similar to the TCP/IP socket model. The sender process performs a PSM Send and the receiving process has to perform a PSM Recv to receive the data. MPI tag and context information is used to perform message matching. Since the communication model requires two processes to be involved this communication model is also colloquially called the two-sided model or the send/receive model.

InfiniPath HCAs do not contain an embedded processor and all protocol operations are performed by the host processor. This additional burden means the host processor is not available for application processing. Our design explores ways by which we can alleviate the load on the host processor by offloading the copy operations specific to one-sided communications.

3.2 Design of Passive Remote Memory Interface

In this section we discuss the challenges and design issues for implementing passive synchronization based MPI one-sided interface over an on-loaded interface. The on-loaded interface only provides a two-sided communication model as described earlier. First we describe the basic data flow of the an MPI instance. Following the example we discuss our designs in more detail.

3.2.1 Overview

Figure 3.1 illustrates the data flow of the remote memory access model.

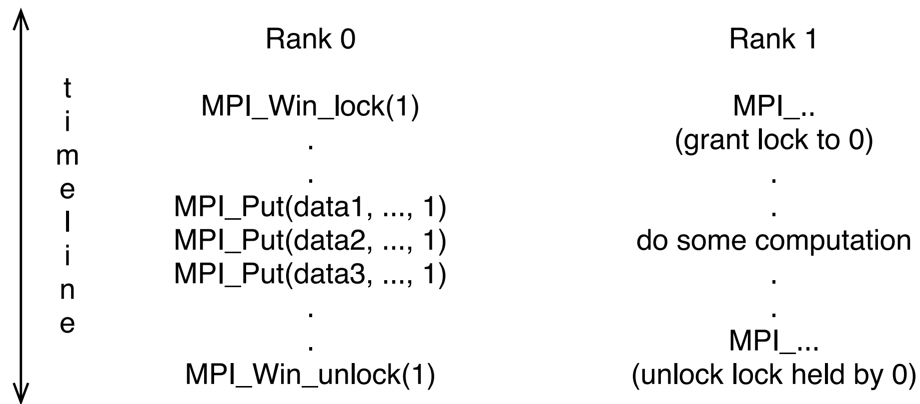

Figure 3.1: Overview of the RMA operations

In the figure, rank 0 acquires a the lock of target rank 1. The target rank is required to be in the MPI library context to completion this operation. Once the lock is acquired, rank 0 can perform all the RMA operations to the target rank.

On completion of the one-sided operations the rank 0 will release the lock using the Win_unlock operation. This illustrates the basic data/communication flow of a RMA operation. Now we consider the designs for implementing the RMA interface on an on-loaded adapter.

3.2.2 Basic Design (BD-RMA):

The basic design for the passive one-sided interface using a two-sided model requires active involvement of the target process. Since some interconnects lack RDMA and remote atomic locking capabilities, designs on such interconnects use a a two-sided protocol. Figure 3.2 shows a one-sided operation being performed from rank 0 to rank 1.

Rank 0 initiates the passive one-sided operations by issuing the window lock operation. A lock message is sent to rank 1 and rank 0 waits for the lock to be granted. Once the lock is granted, rank 0 performs the MPI_Puts followed by the unlocking of the window. MPI semantics ensures that only after all the Puts are complete, the lock is released. All the one-sided messages are received into pre-posted buffers and then copied to the actual location in the target window.

Figure 3.2 clearly shows the problem with this design. The target rank is performing computation and only when it enters the MPI library does it respond to the lock protocol. This causes a large sender latency and also provides zero overlap of computation-communication on the receiver side. We resolve this issue in our second design which uses a multi-threaded approach to ensure the progress. With the emergence of multi-core processors having a dedicated library thread ensure progress is not a infeasible approach.

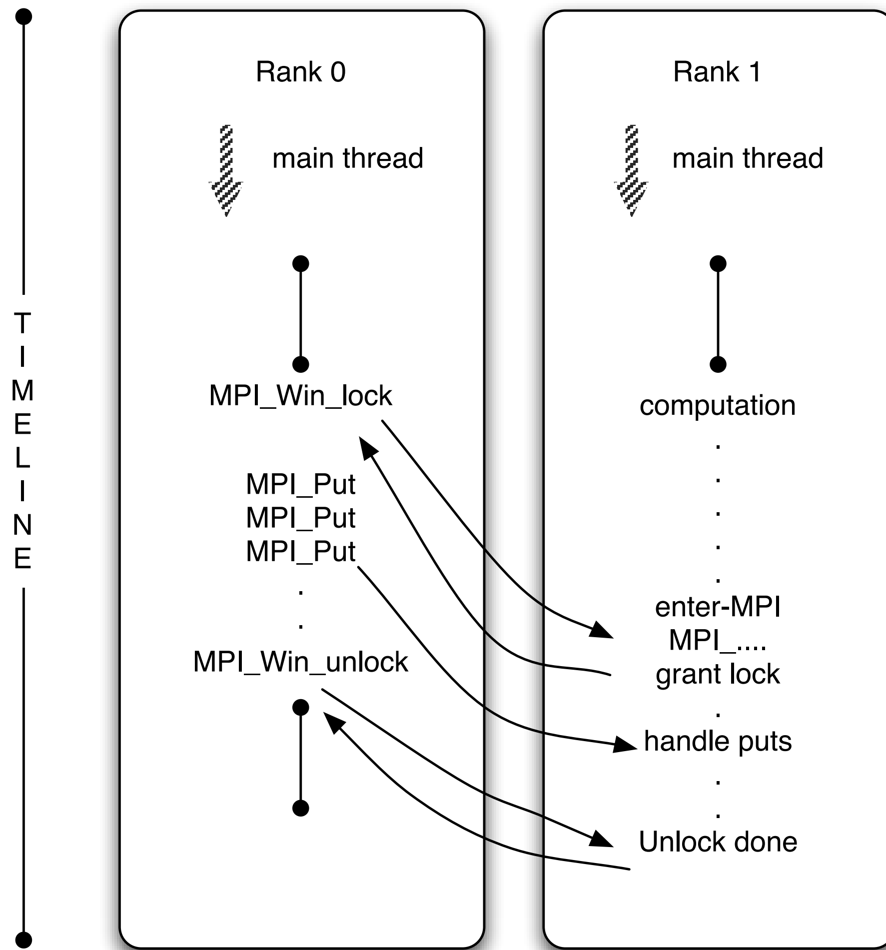

Figure 3.2: Basic RMA Design (BD-RMA)

3.2.3 Helper Thread Design (TH-RMA)

The basic design presented in Section 3.2.2 does not provide any communication-computation overlap and has a high sender overhead. The target rank does not enter the MPI library and hence the origin rank does not progress with the one-sided communication. One solution to alleviate this issue is shown in Figure 3.3.

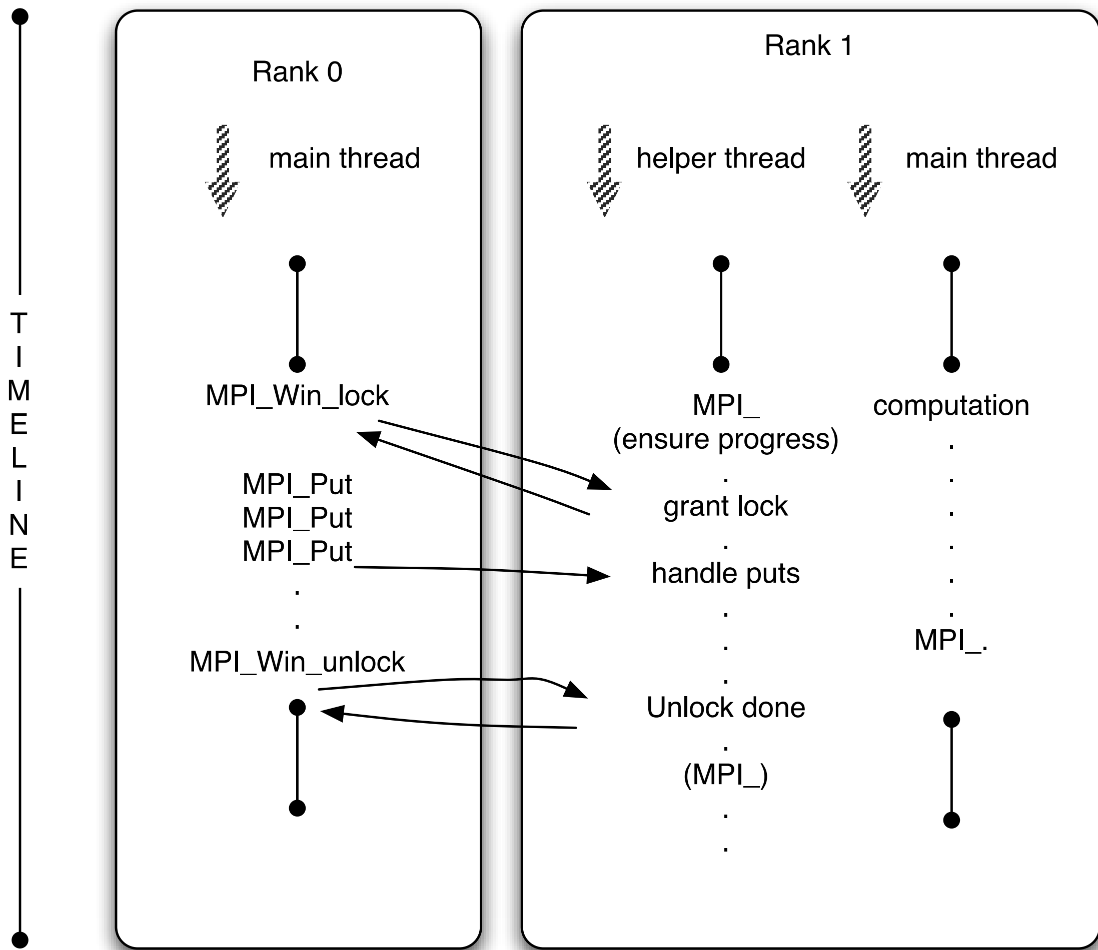

Figure 3.3: Helper Thread Design (TH-RMA)

In this design we have an additional helper thread running in the MPI library. With this design, if the main thread is not in the MPI library, the helper thread can ensure progress. This enables the one-sided communication to progress immediately. Like the basic design, the lock message is rst received into pre-posted bu?ers. But due to helper thread, the lock is granted immediately. Subsequently the Puts and the unlock operations are also handled by the helper thread. The helper thread copies

out the received Puts from pre-posted bu?ers into target memory windows. In our threaded design we also address the following design issues:

- One major demerit of this approach is the CPU contention between the main thread and the helper thread. We resolve this by cancelling the helper thread *pthread_cancel* if the main thread enters the MPI library. When the main thread exits the MPI library the helper thread is re-created. This ensures no lock or CPU contention, the relatively cheap cost of creating a new thread makes this approach feasible.

- The helper thread actively polls for communication progress. This would cause CPU contention with the main thread. To resolve this we keep the completion polling less aggresive in the helper thread. The helper thread sleeps intermittently if no completion event is detected. If a completion event is detected, we aggressively complete the processing of the polled event.

- One issue with this design is that if the amount of data transferred is very large, the helper thread will spend a lot of time performing memory copies. The memory copy is unavoidable without RDMA support. We try to solve this by using the Intel I/O Accelerator technology (IOAT). The IOAT engine provides hardware support for performing memory copies using DMA. Using the I/OAT engine for copying large data also prevents the cache from being polluted thus enabling the foreground computation to proceed without cache effects.

We address the above issue by utlilizing the Intel I/O Acceleration engine to provide hardware copy-offload. The next section introduces the I/OAT device and describes the design of the MPI-2 RMA interface using the I/OAT device.

3.3 Design of Passive Remote Memory Interface using I/OAT

We first provide an overview of the Intel I/OAT technology followed by the design of the copy-offload interface in Section 3.3.2. Section 3.3.3 presents the design of the one-sided interface with I/OAT.

3.3.1 Overview of I/OAT

Intel I/O Acceleration Technology (I/OAT) is an I/O acceleration technology developed by Intel and available in most modern Intel Chipsets. The technology provides a DMA engine to offload data-movement and reduce CPU overhead. On chipsets with this feature, the I/OAT device is a PCI resource with a respective I/OAT DMA driver. The I/OAT DMA engine can be used as copy-offoad engine, allowing the processors to perform useful tasks. As mentioned in [26], using a copy-offload engine provides a reduction in CPU usage and yields better performance. Copy engines can move data in blocks larger than word-size and hence can provide higher performance on larger data sets. Also, since the memory copy progresses asynchronously with computation using the offload engine, it provides copy-communication and also reduces cache pollution.

3.3.2 I/OAT Copy Offload Design

The architecture of the copy offload is shown in Figure 3.4 The copy offoad engine is implemented as a Linux kernel module. User space applications can issue copy

requests to the module via an ioctl call. The kernel queues the requests unless an explicit issue is performed, allowing multiple requests to be batch issued. On issueing a copy, the dma-engine provides a cookie that can be polled for completion.

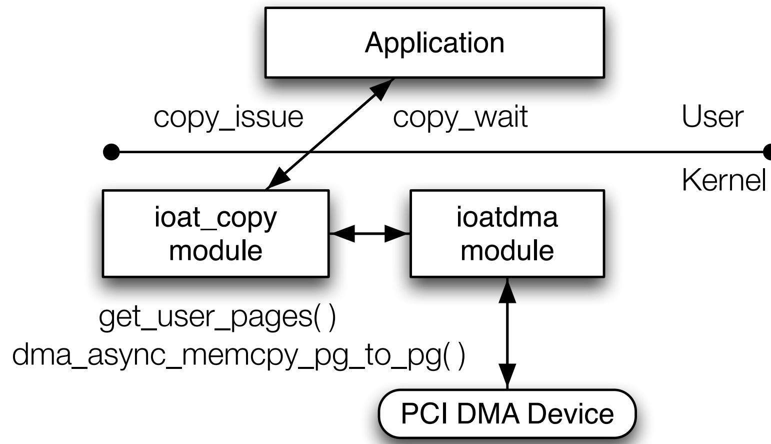

Figure 3.4: I/OAT Copy offload architecture

The *ioat_copy* module locks the user memory space using kernel API *get_user_pages* and issues one DMA operation per page. The I/OAT DMA kernel interface exposes the *dma_async_memcpy_pg_to_pg* interface which issues one copy request per page. The cookies returned by the DMA engine are stored for future polling. Completion of all the issued copies are ensured by a single system call similar to MPI Waitall, which returns only when all outstanding copies are complete.

3.3.3 I/OAT Based Design (IO-RMA)

Figure 3.5 shows the design of the one-sided interface with the I/OAT copy offload.

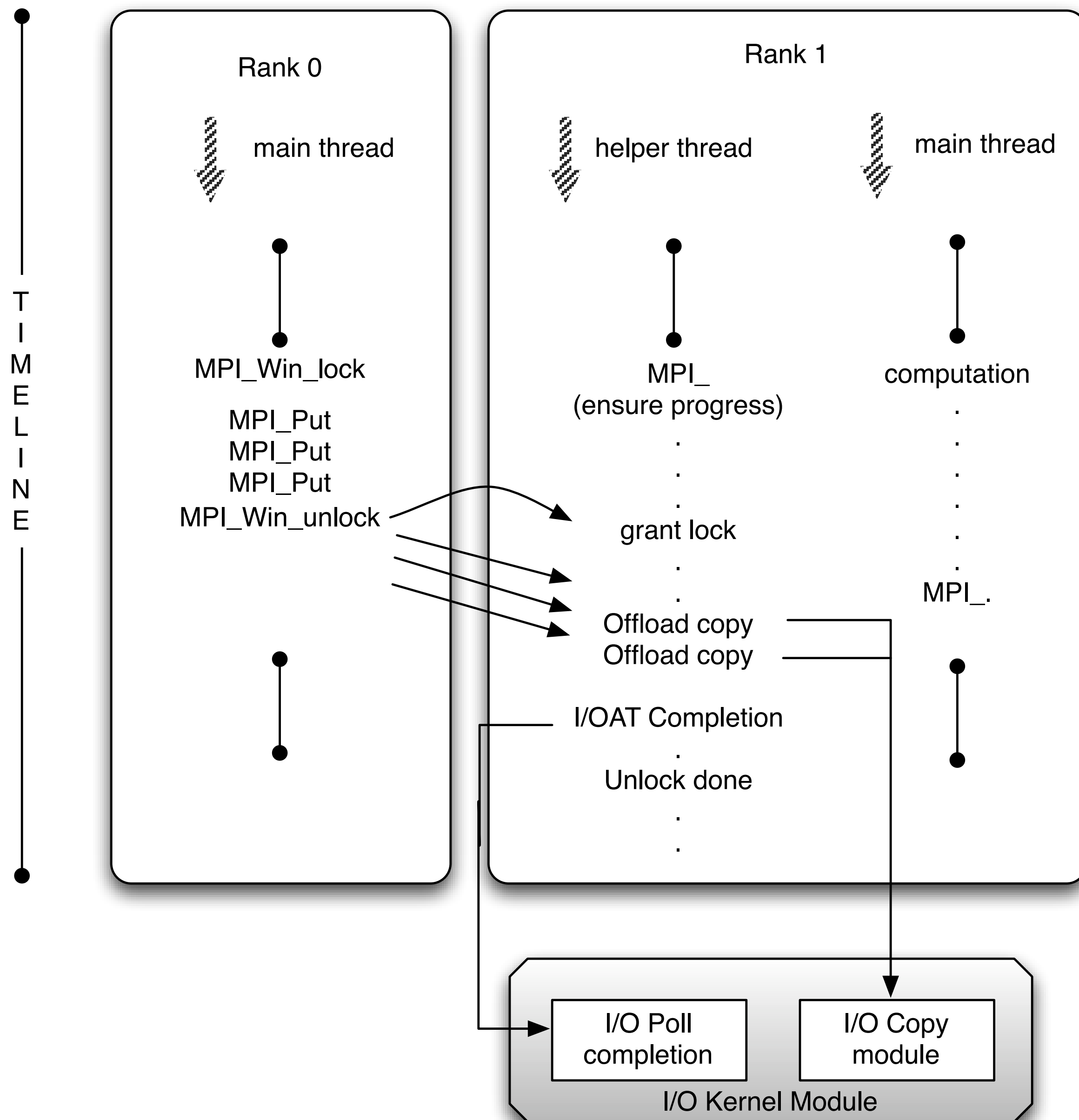

Figure 3.5: I/OAT RMA Offloading design (IO-RMA)

In this design, the one-sided operations are not issued by the origin at the time of the lock operation. Instead the origin rank queues all the issued one-sided requests and delays issue until the unlock. At the unlock stage, the origin process knows the exact size and number of operations in the queue. The lock request now performs two tasks: (i) it requests for a lock from the target rank and (ii) it informs the target rank the exact number of operations it will issue. Prescence of the helper thread enables us to process the lock request immediately. Additionally, the helper thread maintains the count of the incoming one-sided requests. Incoming Puts and Accumulates are issued for memory copy to the I/OAT module. Knowing the count of the one-sided operations allows us to batch multiple DMA issues in a single system call. Once all the one-sided operations from our origin rank is processed, the I/OAT engine is polled for completion and the lock is released.

Currently, I/OAT does not expose an interrupt driven completion semantic. Thus we need to perform active polling to check for completion, however, polling is not required for progress. Since polling is done only to detect completion, the operation is kept lightweight. I/OAT completion semantics provides us the last completed cookie. Since we internally maintain the list of pendings requests we can use this informatio to calculate the number of outstanding DMA requests. If any pending requests exist, the polling is suspended and the thread is put to sleep for a small time interval. Upon waking the thread again checks for pending requests and iterates until it nds all requests to have completed. Once all the DMA requests are completed the lock is released and the origin rank is informed. Since, most of the data movement is handled by the I/OAT engine and completion polling requires only a single system call

by the helper thread, the main thread can proceed with computation with minimal overhead.

3.4 Performance Evaluation

In this section we present the experimental evaluation of our designs. The testbed used is an Intel cluster. Each node is a dual processor (2.33 GHz quad-core) system running an Intel 5000X Chipset with I/OAT support. Each node has 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 InniPath QLE7140 HCAs with PCI Express interfaces. The operating system used is RedHat Linux 2.6.18-92. We implement our designs in the MVAPICH2 [15] library.

3.4.1 I/O AT Micro-benchmarks

In this section we present the basic I/OAT performance gures. Figure 3.6 shows the I/OAT memory copy latencies for small sizes. The I/OAT copy offload has an initial overhead which leads to sub-optimal performance for small copies. However, as seen in Figure 3.7 the performance of the copy offload is better for large data sizes. For data sizes of 4MB and above the I/OAT provides a better latency. We believe this is due to the fact that I/OAT is able to move data in cache-size blocks while memcpy can move data only word size at a time.

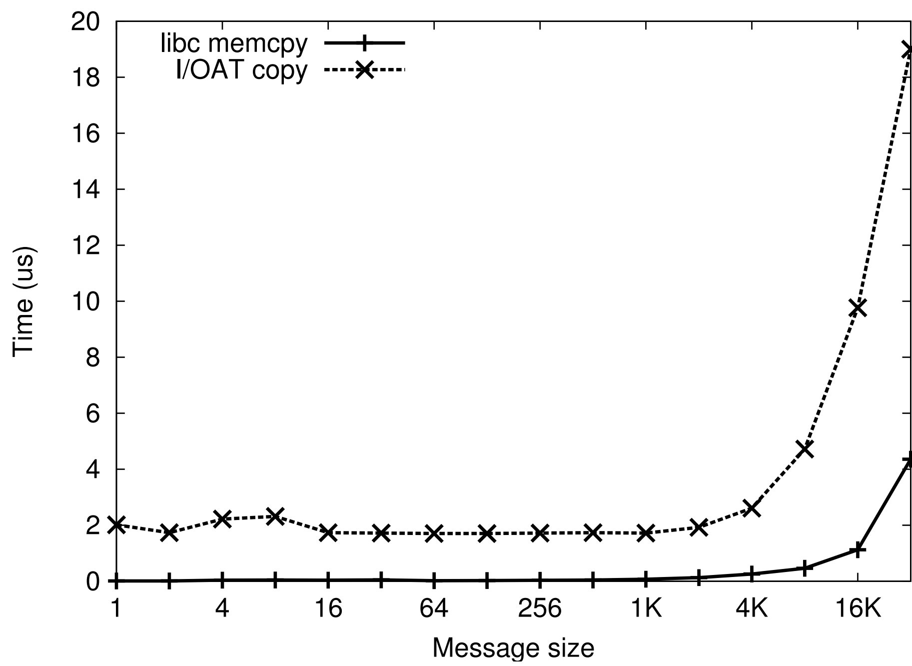

Figure 3.6: Copy Latency (small)

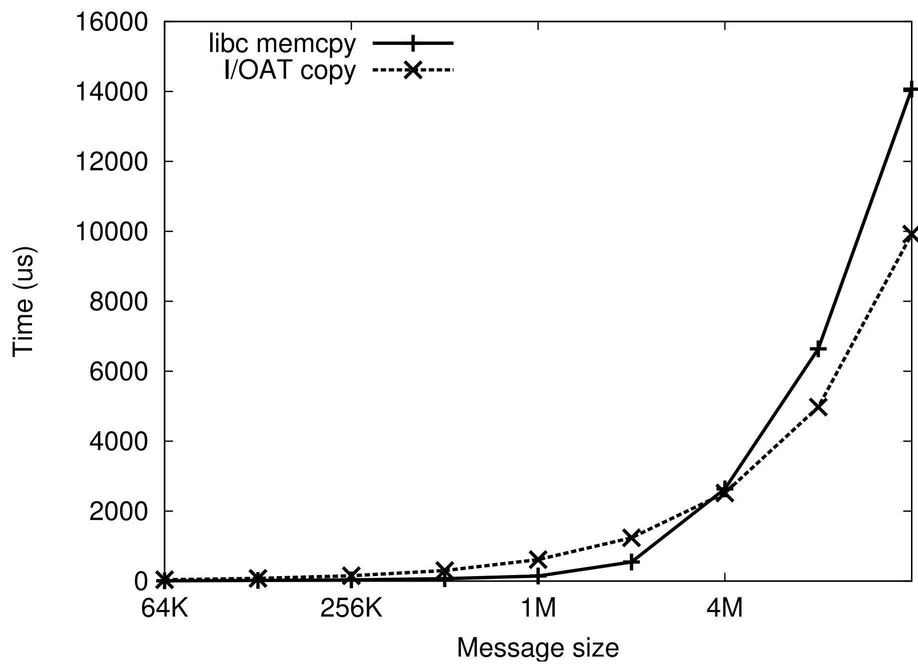

Figure 3.7: Copy Latency (large)

3.4.2 MPI Benchmarks

Figures 3.8 and 3.9 show the MPI_Put latency with the three designs discussed. The MPI_Put latency measurement was done without any computation on the receiver side. The basic design and the design with helper thread perform very similarly. We do not see any signicant overhead incurred due to the thread cancellation in the helper thread based design. With the I/OAT design, small messages suffer from a higher latency due to the higher cost of issueing and completing the DMA request. However, since multiple DMA requests can be issued for a copy the overall latency per Put operation starts to improve with our I/OAT design. For large message sizes, Figure 3.9 clearly shows signicantly lower latencies. At 4MB message sizes, the I/OAT provides upto 30% lower latency than the basic design.

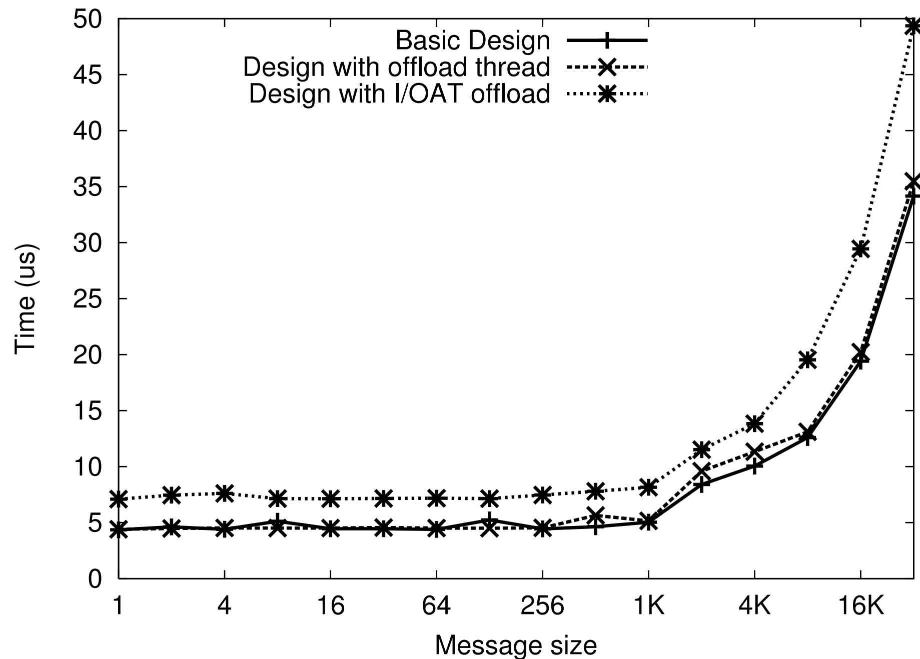

Figure 3.8: MPI_Put Latency (small)

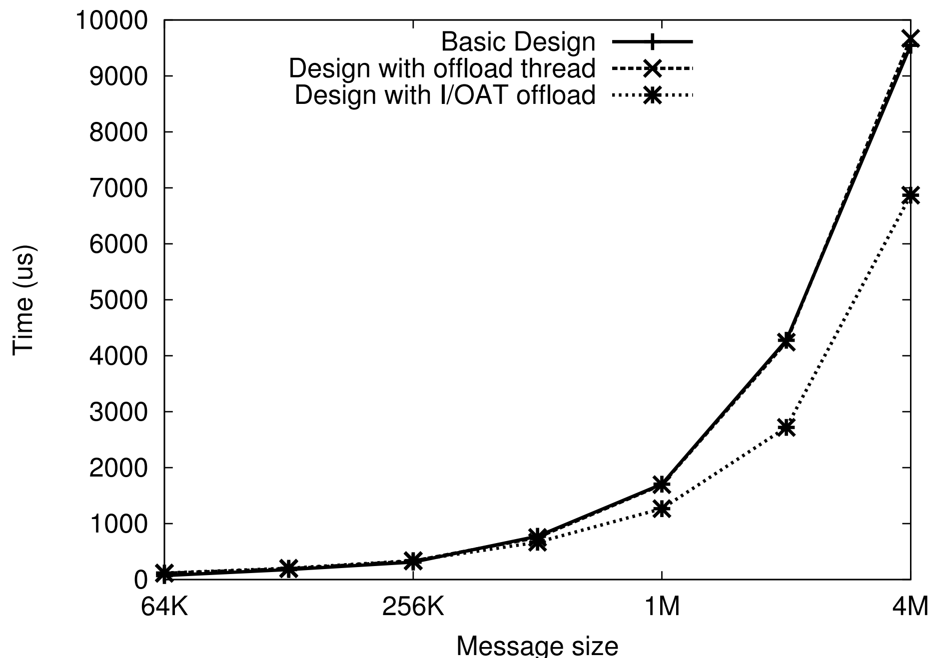

Figure 3.9: MPI_Put Latency (large)

We measure the MPI Put messaging bandwidth using passive synchronization. Figure 3.10 shows the bandwidth achieved by the three designs. For small messages the basic design and the helper thread design perform well. The I/OAT design has a lower bandwidth due to higher overhead incurred due to the system call and the DMA polling. However, for larger messages $> 32k$ we can see the benefits of the new design. The benefits start as slow as 32K because multiple Put operations are concurrently issued for DMA. The I/OAT DMA engine can assure progress of multiple DMA requests and this allows concurrent copy operation corresponding to the Puts, as opposed to the serialized copy of all the Puts in the other two designs.

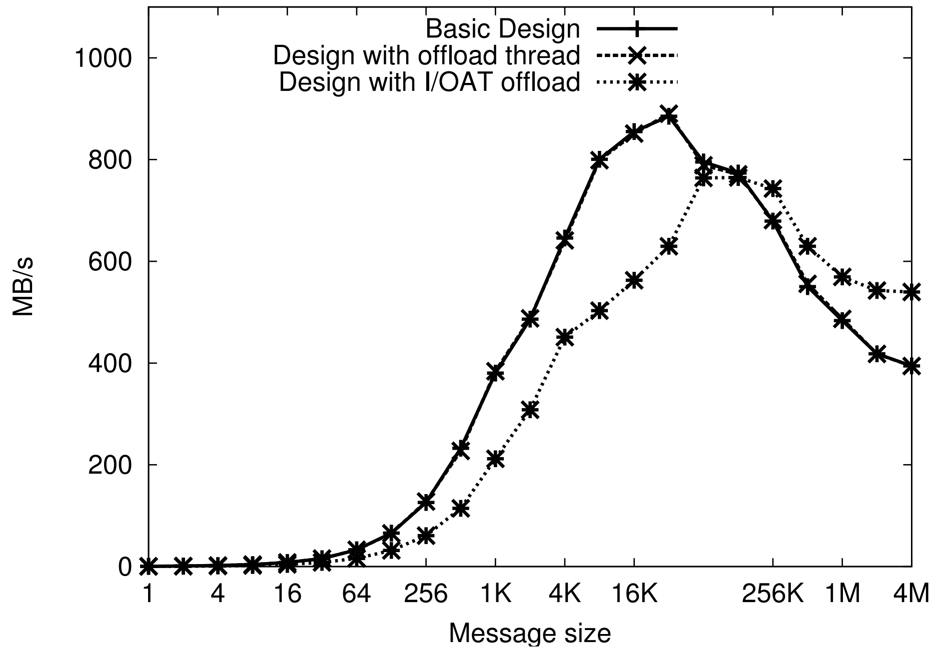

Figure 3.10: MPI_Put Bandwidth

3.4.3 Computation-Communication Overlap

In this section we measure the communication-computation overlap obtained by our designs. In our design, the helper thread is always assigned to the same CPU as the main thread. Using other CPUs in themulti-core systems to run the helper thread is not a realistic solution, hence this limits the overlap achievable. The overlap experiment, considered the time spent by the receiver without computation as the basic communication time. We introduce a comparable amount of computation in the receiver process in the overlap test. If the overall latency of the receiver does not change it signifies a100% overlap of the computation and the communication. As seen in Figure 3.11, with the basic design, there is zero overlap and this is expected.

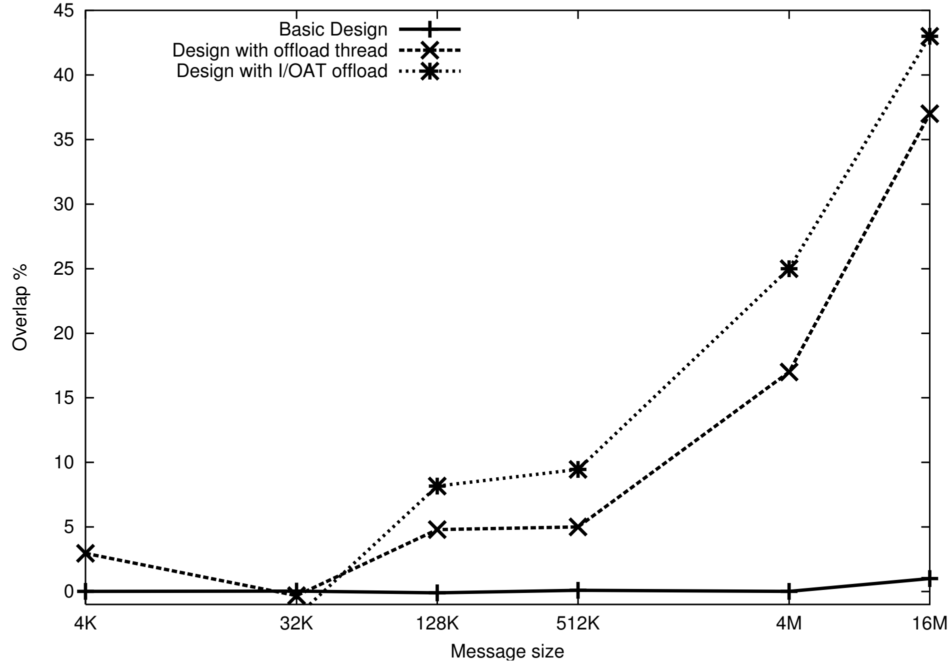

Figure 3.11: Computation-Communication Overlap

With the helper thread based design, due to low transmission time of small messages, the overlap achieved is minimal. But for higher message sizes $> 128K$, because the helper thread ensures progress the lock is acquired quickly and the sender can initiate all the data transmissions. This transmission time is now overlapped with computation time. With the I/OAT based design, the transmission time is overlapped with computation time as in the previous case. Additionally, the copy operations progress concurrently with the computation, thus giving a significantly higher overlap value. We see upto 25% with 4MB message sizes using the I/OAT design and even higher overlaps for 16MB. As the message size increases the transmission time increases significantly (a part of the overlap). The copy time also increases for large messages (providing more overlap), due to this the overlap will keep increasing with

increasing message sizes. The overlap can never be 100% as the same core performs both the computation and the communication progress.

3.4.4 Effect on Caches

To evaluate the effect of our designs on the caches we designed an experiment to mea- sure the L2 caches misses incurred. The experiment performs passive synchronization based one-sided operations to a target rank, while the target rank performs a compu- tation. We measure the L2 cache misses seen by the computation loop. At the end of each loop of the computation the two ranks synchronize and repeat the computation and one-sided communication. We use the oprofile Linux proler to measure the L2 cache misses incurred. The resulting cache misses are averaged over 10 runs.

Table 3.1: L2 Cache misses

Message size	Basic design	Design with helper thread	Design with I/OAT
32K	481625	736833	378500
256K	490000	850000	366000
1M	478875	894244	371500
4M	522000	896300	366500

As seen in Table 3.1 we see that the basic design has a high L2 cache miss rate. Due to the iterative nature of the test we see that the program suffers from the compulsary misses during the computation loop. However, completing the one-sided communication at the synchronization point pollutes the cache and some of the compulsary misses reoccur for next iteration. For this reason, the total L2 misses are very

similar across the range of message sizes, only slightly increasing. The helper thread based design, has a high cache miss rate due to the same L2 cache being shared by two threads. Each thread brings in data to the cache that is evicted by the other thread. The cache miss measurements for this case varies significantly due to the dependence on the scheduling of the two threads. The third column shows the lowest cache miss numbers, for the design with I/OAT offload. The primary reason being I/OAT moves data from memory to memory and hence the CPU caches remain unaffected. The computation loop alone without any communication suffers from a total of 358,000 L2 misses. This experiment clearly demonstrates the efficiency of the I/OAT offload design with respect to CPU caches.

3.5 Related Work

Several MPI-2 implementations support the one-sided communication model, MPICH2 [14], MVAPICH2 [15], OpenMPI [16], are some of the open-source implementations of MPI- 2. Thread based design of passive synchronization have been proposed in [7], however their designs do not solve the CPU contention issue when the main progress thread executes the communication progress. Also, the naive usage of an helper thread, leads to the helper thread consuming CPU even when no one-sided communication exists. Our approach denes the helper thread as a generic progress thread that can progress any communication event seen on the network. Asychronous progress for rendezvous communication using a thread based approach has been studied in [10].

I/OAT feature has been used to achieve asynchronous memory copy in the context of data-centers [24, 22, 23]. The Linux TCP/IP receive stack introduced the

Network DMA feature using I/OAT to reduce server overheads. Our work differs in the aspect that we use I/OAT to offload data movement on the target side in MPI-2 one-sided communication.

3.6 Summary

The one-sided (RMA) communication model in MPI provides one-sided semantics to the application writers. The passive one-sided communication mode can minimize the coordination between the origin and target process and can ideally provide good computation-communication overlap. However, this requires hardware support from the networks in the form of RDMA read/write and remote locking capabilities. When the network cannot provide such capabilities, the implementation is usually done on top of two-sided semantics leading to sub-optimal performance.

Thus efficient designs of the one-sided interface needs to be explored on RDMA-incapable networks. In this paper, we present a common basic design of implementing the one-sided interface over two-sided communications. We extended the basic design by proposing a new helper thread based design to ensure quick communication progress in passive one-sided models and an I/OAT copy offload based design to alleviate CPU consumption. The experimental evaluations showed signicant performance benets for large messages when I/OAT offloading is used. Using the I/OAT engine provided the added benet of keeping the caches unpolluted, a major side-effect in memcpy based designs. In this work we use Infinipath network as a case-study, however the designs presented are generic and are applicable to other RDMA-incapable interconnects (such as naive 1/10 Gigabit Ethernet) and systems with DMA based copy support.

CHAPTER 4

CONCLUSIONS AND FUTURE WORK

In this thesis, we have designed the MPI-2 programming interfaces on InfiniBand interconnects. Our work involved designing effective startup and setup of dynamic MPI processes and the optimization of MPI-2 RMA passive interface on onloaded InfiniBand adapters.

4.1 Designing MPI-2 Dynamic Process Management

Few researchers have addressed designing the MPI-2 dynamic process interface. To the best of our knowledge, this work is one of the first that addresses the design issues in dynamic process interface and provides a detailed design for InfiniBand. We also provided benchmarks and a real-world application as a proof-of-concept .

Our designs achieves the highest spawn rate of know MPI-2 InfiniBand implementations. Additionally, the application suffers from only minimal or no overhead when designed over the dynamic process interface.

In future, we plan to carry out studies into the design of inter-communicator collectives and design more applications using the dynamic process interface.

4.2 Designing MPI-2 Remote Memory Access

In Chapter 3, we presented the existing designs for the MPI-2 RMA interface using passive synchronization. The designs showed that the basic design (BD-RMA) which is used by several MPI implementations does not perform well. It has a high sender overhead and provides no overlap. The thesis proposes a novel design for the RMA interface by using the Intel I/OAT DMA engine to offload data movement overheads. Though we evaluated our implementation on InfiniPath, the two-sided nature of solution can be applied on any interconnect such as Ethernet.

In future, we plan to explore interrupt based designs for reducing CPU consumption by the I/O kernel module.

BIBLIOGRAPHY

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

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

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

[4] Márcia C. Cera, Guilherme P. Pezzi, Elton N. Mathias, Nicolas Maillard, and Philippe Olivier Alexandre Navaux. Improving The Dynamic Creation of Processes in MPI-2. In *EuroPVM/MPI*, pages 247–255, 2006.

[5] QLogic Corporation. http://www.qlogic.com/default.aspx/.

[6] Jack J. Dongarra Edgar Gabriel, Graham E. Fagg. Evaluating Dynamic Communicators and One-Sided Operations for Current MPI Libraries. In *International Journal of High Performance Computing Applications*.

[7] Weihang Jiang et al. Efficient Implementation of MPI-2 Passive One-Sided Communication on Infiniband Clusters.

[8] W. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 530, Washington, DC, USA, 1995. IEEE Computer Society.

[9] Sangbum Kim, Namyoon Woo, and Heon Y. Yeom. Design and Implementation of Dynamic Process Management for Grid-Enabled MPICH.

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

[11] Ewing Lusk. Fault Tolerance in MPI Programs. *Special issue of the Journal High Performance Computing Applications*, 18:363–372, 2002.

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

[13] MPI-Forum. http://www.mpi-forum.org/.

[14] MPICH2. http://www.mcs.anl.gov/research/projects/mpich2/.

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

[16] OpenMPI. http://www.open-mpi.org/.

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

[18] POV-Ray. http://www.povray.org/.

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

[20] J. Sridhar, M. Koop, J. Perkins, and D. K. Panda. ScELA: Scalable and Extensible Launching Architecture for Clusters. In *International Conference in High Performance Computing (HiPC08)*, December 2008.

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

[22] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT. In *International Conference on Cluster Computing*, 2007.

[23] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *CAC*, 2007.

[24] K. Vaidyanathan and D. K. Panda. Benefits of I/O Acceleration Technology (I/OAT) in Clusters. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2007)*, 2007.

[25] Weikuan Yu, Qi Gao, and D.K. Panda. Adaptive connection management for scalable MPI over InfiniBand. *Parallel and Distributed Processing Symposium, International*, 0:81, 2006.

[26] Li Zhao, R. Iyer, S. Makineni, L. Bhuyan, and D. Newell. Hardware Support for Bulk Data Movement in Server Platforms. pages 53–60, Oct. 2005.