Mississippi State University
Scholars Junction

Theses and Dissertations Theses and Dissertations

12-13-2002

Best Effort MPI/RT as an Alternative to MPI: Design and Performance Comparison

Raghavendra Angadi

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation

Angadi, Raghavendra, "Best Effort MPI/RT as an Alternative to MPI: Design and Performance Comparison" (2002). *Theses and Dissertations*. 866.
https://scholarsjunction.msstate.edu/td/866

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

BEST EFFORT MPI/RT AS AN ALTERNATIVE TO MPI:

DESIGN AND PERFORMANCE COMPARISON

By

Raghavendra Angadi

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science

Mississippi State, Mississippi

December 2002

BEST EFFORT MPI/RT AS AN ALTERNATIVE TO MPI:

DESIGN AND PERFORMANCE COMPARISON

By

Raghavendra Angadi

Approved:

Anthony Skjellum
Associate Professor of Computer Science
(Major Professor)

Donna S. Reese
Associate Professor of Computer Science
(Committee Member)

Susan M. Bridges
Professor of Computer Science
(Committee Member)

Julian E. Boggess
Associate Professor of Computer Science
Graduate Coordinator
Department of Computer Science

A. Wayne Bennett
Dean of the College of Engineering

Name: Raghavendra Angadi

Date of Degree: December 13, 2002

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: BEST EFFORT MPI/RT AS AN ALTERNATIVE TO MPI: DESIGN AND PERFORMANCE COMPARISON

Pages in Study: 59

Candidate for Degree of Master of Science

The Real-Time Message Passing Interface (MPI/RT) is an emerging real-time communications middleware standard for distributed real-time applications. The Message Passing Interface (MPI) is the de facto standard for high performance parallel application development. In this thesis, we describe how MPI/RT with best effort quality of service can be used as an alternative for MPI. Mercury Computer Systems' RACE embedded parallel computer is used as the platform for comparison of design and performance of these two standards. The main advantages MPI/RT has over MPI are its explicit support for communication channels and its emphasis on *early binding*. Design and implementation of best effort MPI/RT on Mercury is described and its performance is compared with MPI in order to illustrate how MPI/RT features allow implementations to exploit the underlying platform more optimally. The results for the benchmarks show that MPI/RT outperforms MPI in almost all cases examined.

DEDICATION

To the fond memory of *maa naanna*.

ACKNOWLEDGMENTS

This thesis would never have seen the light of the day without excellent guidance, advice and motivation from Dr. Anthony Skjellum. I thank him for his invaluable support and encouragement. I would like to thank MPI Software Technology, Inc. for providing me an excellent opportunity to take part in implementation of MPI/RT and MPI on Mercury embedded parallel computer. This not only provided the foundation for all the work that is presented in this thesis, it has given me valuable education in real world high performance software development. MPI Software Technology, Inc. has been extremely generous in letting me use the software and hardware that is vital for this thesis. I would like to specifically thank Zhenqian Cui, Dr. Rossen Dimitrov (former MSU students), and Dr. Skjellum with whom I have worked closely on development of this software at MPI Software Technology, Inc. I would like to thank my committee members Dr. Donna Reese, Dr. Little, and Dr. Susan Bridges for their support. I always enjoyed insightful discussions with my fellow HPC lab members. I thank them for making the lab a fun place to work. Finally, I am grateful to Dr. Edward Allen for providing an excellent template that made typesetting this thesis in LaTeX so much easier.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

I. INTRODUCTION 1

1.1 Motivation 1
1.2 Message Passing Standards 2
1.2.1 MPI 3
1.2.2 MPI/RT 3
1.2.3 Best Effort MPI/RT and MPI 5
1.3 Hypothesis and Main Goals 5
1.4 Contributions 6
1.5 Organization 6

II. RELATED RESEARCH AND LITERATURE REVIEW 7

2.1 MPI 7
2.1.1 Basic Model 7
2.1.2 Persistent Communication 9
2.1.3 MPI-2 9
2.2 MPI/RT 10
2.2.1 Structure of an MPI/RT application 11
2.2.2 Communication Primitives 12
2.2.2.1 Buffer 13
2.2.2.2 Buffer Iterators 13
2.2.2.3 Channels 14

CHAPTER Page

III. DESIGN OF MPI/RT AND MPI ON MERCURY 17

3.1 Mercury Embedded Multicomputer 17
3.1.1 Overview 17
3.1.2 Shared Memory Buffers 18
3.1.2.1 Programmed I/O 18
3.1.2.2 Direct Memory Access 19
3.1.3 DX Transfer Creation 19
3.2 MPI/RT Communication Subsystem 20
3.2.1 Buffers 21
3.2.2 Channels 21
3.2.3 Committing a Ptchannel 23
3.2.4 MPI/RT Data Transfer Protocol 25
3.3 MPI Communication Subsystem 26
3.3.1 DMA Transfers 26
3.3.2 MPI Data Transfer Protocol 27
3.3.2.1 Synchronization and Flow Control in MPI 27
3.3.2.2 The Transfer Protocols 28
3.4 Summary of Design Differences 30

IV. RESEARCH METHODOLOGY AND EXPERIMENTS 31

4.1 Latency and Bandwidth Measurements 31
4.2 3-D Poisson Solver 32
4.3 RT_Cornerturn 32
4.4 An image processing example 34

V. RESULTS AND ANALYSIS 35

5.1 Experimental Setup 35
5.2 Latency 36
5.3 3-D Poisson Solver 39
5.4 RT_Conrnerturn 42
5.5 Slab – An Image Processing Example 45
5.6 Summary of Results 46

VI. CONCLUSIONS 48

6.1 MPI vs. MPI/RT 48
6.2 Future Work 50

Page

REFERENCES 52

APPENDIX

COMPREHENSIVE SET OF RESULTS 54

LIST OF TABLES

TABLE Page

5.1 Round trip latency for MPI and MPI/RT 38

A.1 3-D Poisson Solver Results I 55

A.2 3-D Poisson Solver Results II 56

A.3 RT_Cornernturn Results I 56

A.4 RT_Cornernturn Results II 57

A.5 Slab Results I 57

A.6 Slab Results II 58

A.7 Slab Results III 58

A.8 Latency and Bandwidth Measurements 59

LIST OF FIGURES

FIGURE Page

2.1 Life cycle of an MPI/RT application 11

2.2 Primary steps during a message transfer on a an MPI/RT Channel 15

3.1 A DX transfer creation in two stages. 20

3.2 Data movement in some of the collective channels in MPI/RT. 22

4.1 Data movement in an all-to-all communication 33

5.1 Round trip latency for MPI and MPI/RT at various data sizes 37

5.2 Bandwidth for MPI and MPI/RT at various data sizes 38

5.3 3D Poisson Solver with 4 Nodes 40

5.4 3-D Poisson Solver with 16 nodes 41

5.5 RT_Cornerturn performance with 4 processes 42

5.6 RT_Cornerturn performance with 16 processes 43

5.7 Scalability of RT_Cornerturn with matrix size 96 44

5.8 Performance of Slab with 4 processes 45

5.9 Performance of Slab with 16 processes 46

CHAPTER I

INTRODUCTION

1.1 Motivation

The Real-time Message Passing Interface [14] is an emerging standard for real-time parallel and distributed computing. Though the primary focus of the standard is to provide a portable API for real-time message passing applications, there are many features in the standard that provide greater scope for library optimizations even for non-real-time (best effort) applications.

We have implemented the MPI/RT standard with best effort quality of service (QoS) at MPI Software Technology, Inc. on Mercury's RACE embedded parallel computer[1]. We have also ported MPI Software Technology, Inc.'s MPI [18] implementation to the Mercury RACE platform[2]. The main motivation for this thesis work is to demonstrate, using the above mentioned implementations, how MPI/RT can better exploit the platform specific features thus providing better performance when compared to MPI on this platform. We would like to encourage best effort MPI/RT as an alternative to MPI by showing that

[1]MPI/RT implementation was developed by Zhenqian Cui and the author.

[2]MPI communication layer for Mercury was initially developed by the author and was later enhanced by Zhenqian Cui. MPI Software Technology, Inc. graciously allowed us to use these middleware libraries as well as the hardware as the basis for this thesis.

many MPI applications can be ported to MPI/RT even though programming models and semantics differ considerably between the two standards. We also discuss various issues that arise during the development of MPI/RT versions of MPI applications because of these differences.

1.2 Message Passing Standards

In the message passing model of parallel computing, an application consists of a set of processing nodes each of which has access to its local memory and communicates with the other nodes by exchanging messages between them. Since the actual mechanism for sending and receiving messages depends on the particular hardware platform, porting parallel applications developed for one class of parallel computers to another is not trivial. In order to develop a parallel application for a specific platform, the developer needs to gain at least minimal expertise in the specific platform. Over the last decade a number of standards for developing parallel applications have emerged in order to address these concerns. Such a standard, usually implemented as a middleware library, often achieves portability and ease of programming with little loss of performance (when compared to the programs developed on the native platform). This has been one of the primary reasons for industry-wide acceptance of such standards. MPI is the most popular among these standards and has emerged as the de facto standard for parallel and distributed application development.

1.2.1 MPI

MPI [12] is an application programming interface (API) for message passing parallel applications. In a message passing model of parallel computing, any processing unit accesses local memory of any other processing unit by explicit message passing. Some examples of processing units are as follows: processes or threads in UNIX, processors on a parallel computer etc. In this thesis, a *process* or a *node* implies a processing unit in a parallel application unless specified otherwise.

An MPI application consists of N processes, each of which is assigned a unique *rank* ranging from 0 through $N-1$. MPI provides operations for both point-to-point communication and collective communication (*e.g.*, broadcast and gather) between these processes. In addition to these basic facilities, MPI provides many other *abstractions* that are commonly used in high performance applications and libraries. Some of the examples are its ability to create subgroups and support for virtual topologies [8].

MPI has contributed enormously to the spread of parallel and distributed computing. Implementations of MPI are now available for most of the commonly used computer systems. Since MPI is a well known standard, in this thesis only the relevant features are described briefly whenever necessary, for completeness.

1.2.2 MPI/RT

In the past, many real-time applications have been developed on “bare” hardware without any kind of operating system [3]. Though possible, it is neither convenient nor easy to

maintain such systems. The problem becomes much more complex when these systems involve more than one node with real-time communication between these nodes. In recent years, many real-time operating systems have been developed that take advantage of distributed nodes [9, 10]. In these systems, the real-time communication subsystem assumes a central role.

Experience with MPI has clearly shown the advantages of having the standard for reducing complexity of developing parallel and distributed applications. Real-Time Message Passing Standard (MPI/RT) aims to achieve similar goals in real-time parallel and distributed application development. MPI/RT is one of the first standards that addresses a wide range of real-time programming paradigms. It supports time-based, event-based, and priority-based models of real-time computing. Relevant features of MPI/RT are described in more detail in later chapters.

In order to provide real-time guarantees, MPI/RT requires that most of the resource requirements of an application be specified before the real-time phase begins. In other words, all the MPI/RT *objects* (which specify resource and quality of service requirements) are *committed* a priori. The encouragement for *early binding* is one of the primary differences between the MPI/RT and the MPI programming paradigms. MPI primarily supports *late binding* (*e.g.*, the parameters for communication are specified only when the the actual communication is initiated).

1.2.3 Best Effort MPI/RT and MPI

All MPI applications can be categorized as applications providing best effort quality of service (QoS). Here, the focus is on achieving high performance by reducing communication overhead and increasing parallelism. Since MPI/RT supports best effort QoS, it is possible to provide MPI/RT versions (with best effort QoS) of many MPI applications. The main advantage of MPI/RT over MPI for these applications is its ability to exploit early binding. In this thesis we are primarily concerned with the performance advantages provided by MPI/RT with best effort QoS over equivalent applications using MPI.

1.3 Hypothesis and Main Goals

The primary goals of this thesis work are as follows:

- to show the applicability of best effort MPI/RT as an alternative standard for developing high performance parallel and distributed applications.
- to show how many of the features in MPI/RT can be exploited effectively to achieve better performance for non real-time applications when compared to the performance of similar applications using traditional message passing standards such as MPI.

Our main hypothesis is that MPI/RT semantics allow better optimizations and better exploitation of the underlying platform, thus achieving better performance for best effort QoS applications when compared to the performance of corresponding MPI applications. We would like to validate the hypothesis through performance analysis of a set of benchmark applications developed using MPI/RT as well as MPI and targeted to the Mercury RACE parallel computer.

1.4 Contributions

The main contributions of this thesis are as follows:

- This thesis presents the first known performance comparison of benchmarks on MPI and MPI/RT.
- This thesis explores the influence of semantics of these two message passing standards on their design on the Mercury RACE embedded platform. The differences in design illustrate how emphasis on early binding semantics in the MPI/RT standard results in optimizations that benefit non-real-time applications as well.
- This thesis presents the design of the first known implementation of a best effort QoS subset of the MPI/RT standard on any platform. This also helped as a prototype implementation during the later developments of the standard.

1.5 Organization

The rest of the chapters are organized as follows: Chapter 2 summarizes MPI/RT and MPI and traces the recent developments. Chapter 3 is a overview of the design of the MPI/RT and MPI communication subsystems with emphasis on how the semantics of these standards affect the design. Chapter 4 outlines the benchmarks used and experiments conducted in order to validate the hypothesis. Chapter 5 discusses the results of these experiments and analyzes the factors affecting the results. Finally, chapter 6 presents basic conclusions, future directions, and some thoughts on MPI/RT as an alternative to MPI for parallel application development.

CHAPTER II

RELATED RESEARCH AND LITERATURE REVIEW

In this chapter, recent developments in MPI and MPI/RT are summarized. A summary of the basics of these standards is also presented.

2.1 MPI

The Message Passing Interface standard is a result of efforts to standardize message passing programming interface for parallel and distributed computing during the early 1990s. An introduction to the standard is provided in section 1.2. A portable public domain implementation of MPI, called MPICH [7], has also contributed to the wide spread acceptance of the standard.

2.1.1 Basic Model

An MPI application consists of N communicating processes (section 1.2.1). MPI is often described as *both* large and small [8]. MPI 1.2 consists of a total of 128 functions but one can write moderately large parallel applications using just six of the the functions. A brief description of these six functions provides a good overview (for brevity, only the essential arguments for the functions are shown):

`MPI_Init()`
: Initializes the library. This should be the first MPI function that is called.

`MPI_Comm_size(size)`
: Gives the total number of processes participating in the application.

`MPI_Comm_rank(rank)`
: Gives the rank of the calling calling process.

`MPI_Send(buffer, buf_size, data_type, dest_rank)`
: Sends a data buffer of length `buf_size` located at the address `buffer` to the process whose rank is `dest_rank`.

`MPI_Recv(buffer, buf_size, data_type, source_rank)`
: Similar to `MPI_Send()` which receives the data from another process.

`MPI_Finalize()`
: The final MPI function to clean up and finalize the library.

All messages that are sent have a *tag* associated with them so that these messages can be identified on the receiving side. It can also be noted that for send and receive operations, the application provides the address of the data and the communicating node's rank as the arguments to the operation. The MPI library does not have prior knowledge of the location of the data and the communicating node. These *late binding* semantics of MPI preclude some of the optimizations possible on many platforms.

MPI supports both blocking and non-blocking modes of communication. MPI also has other modes of communication such as *synchronous* and *ready* modes. MPI has explicit support for collective communication operations such as broadcast, scatter and gather. It also makes collective communication much more flexible through *communicators*. Communicators provide a way of forming subgroups of processes.

2.1.2 Persistent Communication

MPI provides limited support for early binding through persistent communication, which can be used when repeated calls to `MPI_Send()` or `MPI_Recv()` with the same arguments are made. Though it is possible to optimize communication in this case by binding the list of arguments, it does not provide full functionality of a conventional channel [12]. MPI persistent communication does not bind both sending and receiving sides. It mainly reduces the overhead of argument passing and argument checking. In practice, few implementations of MPI actually are optimized for persistent communication. Little emphasis is placed on this mode of communication in MPI. On the other hand, channels form the core of the MPI/RT communication model.

2.1.3 MPI-2

As experience with MPI has increased, many researchers have proposed extensions to the standard [17]. These include extensions that improve the performance and scalability as well as support for more models of parallel programming. MPI-2 [13] adds a number of new features in addition to extending the existing functionality. MPI 1.2 (as well as MPI/RT 1.0) supports the static process model where the number of processes participating does not change. MPI-2 provides dynamic process management where new processes can be added or spawned. The standard includes an API for parallel file I/O and data distribution among the processes. MPI-2 introduces *one sided* communication where communication is explicitly initiated only on one side. However, there is little change in

support for early binding or channels in MPI-2. Prototype implementations of the standard are under development at various institutions, and a few commercial implementations are nearly completed as of now.

2.2 MPI/RT

MPI/RT is an emerging standard for a wide range of distributed real-time applications. The main goal of MPI/RT is to provide message-passing functionality with quality-of-service (QoS). MPI/RT supports time-driven, event-driven, and priority-driven models of real-time applications. The wide functionality and the real-time guarantees provided by MPI/RT pose "middle out" requirements on the system; it has a strong influence on the design of both the underlying platform and the applications[1] . At present, there are few operating systems and platforms which can support a complete MPI/RT implementation. A time based real-time kernel based on RT-Linux called TURTLE [1] has been developed at Mississippi State University. The feasibility of basic MPI/RT communication channels with time-driven QoS has been successfully demonstrated [15] using this kernel with Myrinet [4] as the underlying physical network.

MPI/RT also supports best effort QoS. In a best effort QoS application, no real-time guarantees are provided. High performance applications, such as MPI applications, can be considered to be best effort QoS applications. An MPI/RT implementation with best effort QoS has been developed for the Mercury RACE embedded parallel computer at MPI

[1]The idea of *middle out requirements* was first popularized by Dr. Vijay Madisetti

Software Technology, Inc. In order to validate our hypothesis, we describe the design of MPI/RT and MPI on Mercury and compare the performance of the two standards on this platform.

2.2.1 Structure of an MPI/RT application

In order to provide predictable performance and QoS guarantees, it is essential for a system like MPI/RT to have advance knowledge of the application requirements during the real-time phase. As shown in figure 2.1, an MPI/RT application primarily consists of a real-time inner loop in between non real-time stages.

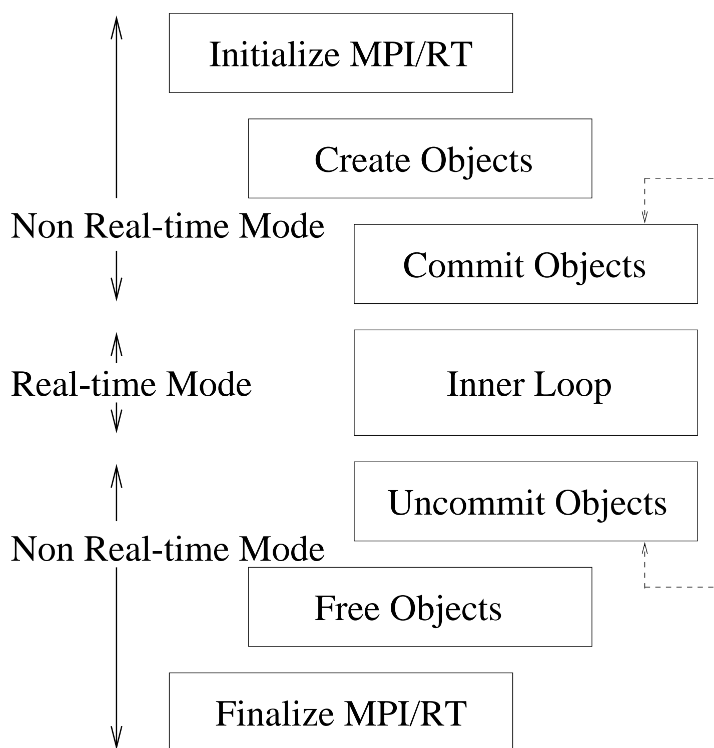

Figure 2.1 Life cycle of an MPI/RT application

After initializing the MPI/RT system using `MPIRT_Init()`, the application creates all necessary MPI/RT objects (*e.g.*, communication channels, buffers and QoS objects). These objects are then *committed* using `MPIRT_Commit()`. During this operation the system learns about the application requirements of various resources such as CPU time, network bandwidth and the QoS guarantees. MPI/RT allocates the required resources during this stage. An admission test to verify if it is possible to provide the required QoS is also performed during this stage. If the `Commit` operation succeeds, the application enters the real-time mode ("Inner Loop" in figure 2.1). In the real-time mode, only a subset of MPI/RT operations are allowed. For example, the application cannot create new objects or change the properties of the existing objects during this stage. The application enters the non real-time mode by invoking `MPIRT_Uncommit()`. During `Uncommit` the system deallocates the resources created during the commit stage. `MPIRT_Finalize()` cleans up the library and ends the MPI/RT mode. The structure of a best effort MPI/RT application does not differ from that of a real-time application, except that QoS arguments are defaulted.

2.2.2 Communication Primitives

This section describes the essential objects that form core of the MPI/RT communication layer.

2.2.2.1 Buffer

A buffer is an object that holds the data that is used in message passing between the nodes. The buffer space for the data can be either supplied by the application or the application can request that MPI/RT allocate the space during the `Commit` operation. Allowing the system to allocate buffer space makes platform specific optimizations possible. For example, a system might allocate buffer space from a region where the network device can directly copy from in order to save an extra copy. After commit, the application can obtain a pointer to the buffer data by invoking `Get_base()` operation on the buffer object.

2.2.2.2 Buffer Iterators

A buffer iterator holds the buffers that are inserted into it. The buffers can be taken from the iterator through `Remove()` operation. One of the parameters of a buffer iterator is a policy that defines the order in which the buffers are placed for removal. The four basic policies are as follows:

1. `MPIRT_BUFITER_FIFO`: First in first out. The buffers are taken from the iterator in the order they are inserted.
2. `MPIRT_BUFITER_LIFO`: Last in first out.
3. `MPIRT_BUFITER_SORTED`: The buffers are ordered from the lowest label to the highest label. The application can specify a label for each buffer.
4. `MPIRT_BUFITER_UNORDERED`: No particular oder specified.

Each buffer iterator can have a set of *allowed buffers* associated with it. Only the buffers that are part of the allowed buffers set are allowed to be inserted into the iterator. If no set

is defined, then any buffer is allowed to be inserted (and removed). All the buffers that are inserted into an iterator have the same size and data type.

2.2.2.3 Channels

A *channel* is a persistent virtual connection between sending and receiving nodes. A channel object primarily consists of the peer node(s) (the nodes participating in the communication), buffer iterators and QoS associated with the channel. A channel has an input buffer iterator and an output buffer iterator. When MPI/RT performs a data transfer over a channel, it removes a buffer out of the input buffer iterator and uses the buffer in the communication. When the communication is completed, it inserts the buffer into the output buffer iterator. There are two modes of activating data transfers over a channel: `MPIRT_Channel_start()` and `MPIRT_Channel_activate()`.

`MPIRT_Channel_start()` initiates one data transfer. This operation is non-blocking. The message transfer takes place according to the QoS specified. For best effort QoS, MPI/RT tries to achieve minimum latency and maximum bandwidth thus completing the transfer as soon as possible. `MPIRT_Channel_Wait()` on a *started* channel blocks until the last initiated transfer (over this channel) is completed.

`MPIRT_Channel_activate()` changes the state to the *activated* where the data transfers take place according to the QoS without explicit calls each time. For example, transfer takes place automatically once in every 10 millisecond window of time over a real-time channel if the QoS specifies a 10 millisecond period. The application needs to

make sure that buffers are available in the input buffer iterator before the start of each period.

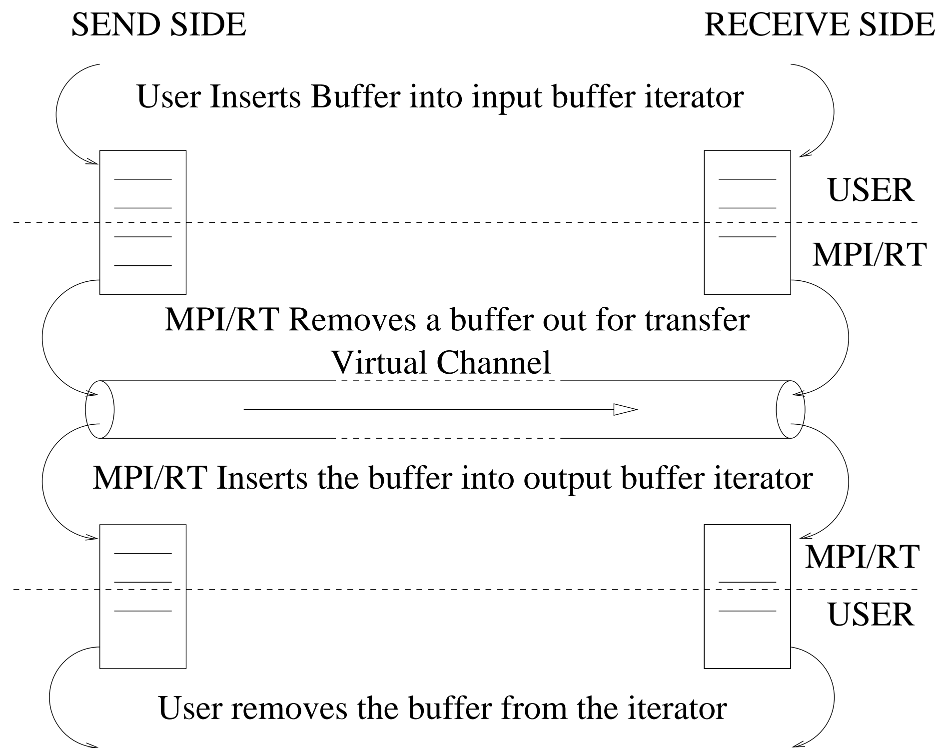

Figure 2.2 Primary steps during a message transfer on a an MPI/RT Channel

The primary steps in a message transfer over a channel are shown in the figure 2.2. The application makes sure that the input buffer iterator is not empty by inserting buffers if necessary. MPI/RT takes one buffer out of the iterator and uses it in the data transfer. The send-side of the channel sends the data in the buffer and the receive-side of the channel fills the buffer. After the transfer is finished, MPI/RT inserts the buffer into the output buffer iterator which can be used by the application.

MPI/RT supports collective channels and subgroups of processes for collective communication. The design of the MPI/RT implementation on the Mercury RACE platform is described in the next chapter.

CHAPTER III

DESIGN OF MPI/RT AND MPI ON MERCURY

In this chapter we describe the design of relevant parts of MPI/RT and MPI communication layer over the Mercury RACE platform. Here, we also present a brief description of Mercury RACE system [11]. Basic knowledge of relevant features of the underlying platform is necessary in order to understand how one middleware might better exploit some of the capabilities of the platform, over the other.

3.1 Mercury Embedded Multicomputer

In this section as well as in this thesis "Mercury" refers to the complete Mercury RACE embedded multicomputer including the hardware as well as its operating system.

3.1.1 Overview

The basic building block of Mercury is its *compute environment* (CE). Each CE consists of a processor, memory, timers, interrupt control, and a DMA controller. Each of the CEs runs Mercury's POSIX like embedded operating system MC/OS. Usually there is more than one CE on a motherboard. These CEs are connected by RACEway which is a network of crossbar switches. RACEway connections give CEs on the same motherboard access to the other CEs' memory at almost the same bandwidth as local memory. Each CE

can run more than one program on it. Mercury is connected to a *host machine* which is usually desktop such as Sun machine running Solaris. Any CE can be directly controlled from the host machine. A process running on a CE can also spawn processes on other CEs.

3.1.2 Shared Memory Buffers

A shared memory buffer (SMB) defines a random access storage block [11]. For a process it does not look different from dynamic memory allocated through `malloc()`. The primary difference is that SMB is can be accessible by processes on the other CEs though the RACEway network. Thus, all the data involved in communication between the CEs is stored in SMBs. An SMB can also be allocated from memory on the devices such as frame grabber that are attached to a CE. Coarse-gain virtual memory management and large page sizes are evidently used with SMBs.

A process can create an SMB from its local memory (using `smb_create()` system call) or it can obtain a *handle* for a remote SMB by attaching to it. A remote SMB can be accessed in one of two ways: Programmed I/O or Direct Memory Access (DMA).

3.1.2.1 Programmed I/O

A remote SMB can be mapped into a process address space using `smb_map()`. After mapping, a process can access the area just as it accesses any other memory. Since each mapping takes up page registers, which are limited, Mercury has a provision to *group* a set

of SMB mappings together. When we actually want to access the SMB, the corresponding mapping can be enabled using `smb_enable`.

3.1.2.2 Direct Memory Access

Mercury provides a device independent interface called DX for high-speed data transfers over RACEway. On the machine where we have conducted our experiments, DX uses the Direct Memory Access (DMA) engine on each CE for this purpose. An application can create a DMA transfer object (or a handle) between a local SMB and a remote SMB and this handle can be used for subsequent DMA transfers.

For large data transfers, DMA is more efficient and for small data transfers (such as 4 or 8 byte transfers), programmed I/O is more efficient.

3.1.3 DX Transfer Creation

A DX (or DMA) transfer object is created in two stages. First, a `DX_template` is created by specifying source and destination SMBs. This operation establishes the path between the source and the destination and usually consumes a considerable amount of resources and time. In the second stage, the amount of actual data transfer and the offsets from the beginning of the SMBs are specified for creating a `DX_transfer` object. Figure 3.1 shows creation of a `DX_transfer` object. If another `DX_transfer` object for the same SMBs with different offsets is required, we need not create another `DX_template` object. Creating a `DX_transfer` object consumes less vital system resources than creating

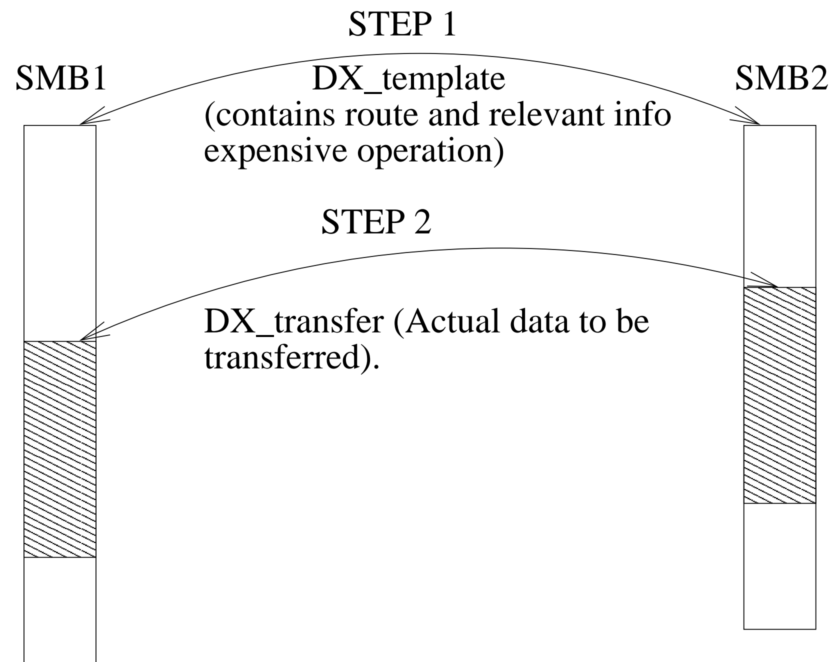

Figure 3.1 A DX transfer creation in two stages.

a DX_template object. In order to achieve scalability, we need to minimize the number of SMBs created (or attached) and the number of DX_template objects created. The next section explains how MPI/RT semantics allow such an optimization of resource consumption. Currently, the offset for a DX transfer object should be a multiple of 8 bytes.

3.2 MPI/RT Communication Subsystem

In this section, we outline the design of MPI/RT channels on Mercury.

3.2.1 Buffers

On each of the processes, only one large SMB for all MPI/RT buffers is created. During the commit stage (section 2.2.1) of an MPI/RT application, the system (MPI/RT library) has complete knowledge of all the buffers that are created by the application. So the system creates one large SMB sufficient for all the buffers and allocates memory from this. Since each of the processes creates only one SMB, each process in an MPI/RT application needs to attach to at most one remote SMB located at the other processes. As there is a limit on the number of SMBs that can be created or attached, this scheme minimally impacts the scalability of the system. Similarly there needs to be at most one `DX_template` object (section 3.1.3) between two given nodes. We observed that the limit on the number of DX template objects that can be created is more stringent than that on the number of SMBs that can be attached.

3.2.2 Channels

An MPI/RT channel (section 2.2.2.3) is a virtual connection between the nodes used for data transfer. MPI/RT supports the following kinds of channels (Figure 3.2), which are patterned after MPI collectives:

`MPIRT_Ptchannel`: Point-to-point channel consists of one send-side and one receive side.

`MPIRT_Bcast_channel`: Root node sends data to all the nodes (including itself) in the group.

`MPIRT_Scatter_channel`: Root sends portions of its data to each of the nodes in the group. The section of the buffer a node receives depends on its rank in the group.

MPIRT_Gather_channel: Similar to a scatter channel except that the root receives the sections of the buffer from each of the nodes in the group.

MPIRT_Reduce_channel: The root gathers data from each of the nodes and performs an operation (such as sum or maximum) on the data from each of the nodes and stores the result.

MPIRT_Barrier_channel: Does not involve any explicit data transfer. When MPIRT_Channel_wait operation (section 2.2.2.3) returns after MPIRT_Channel_start, it implies that all the nodes in the group have executed MPIRT_Channel_start operation on this channel.

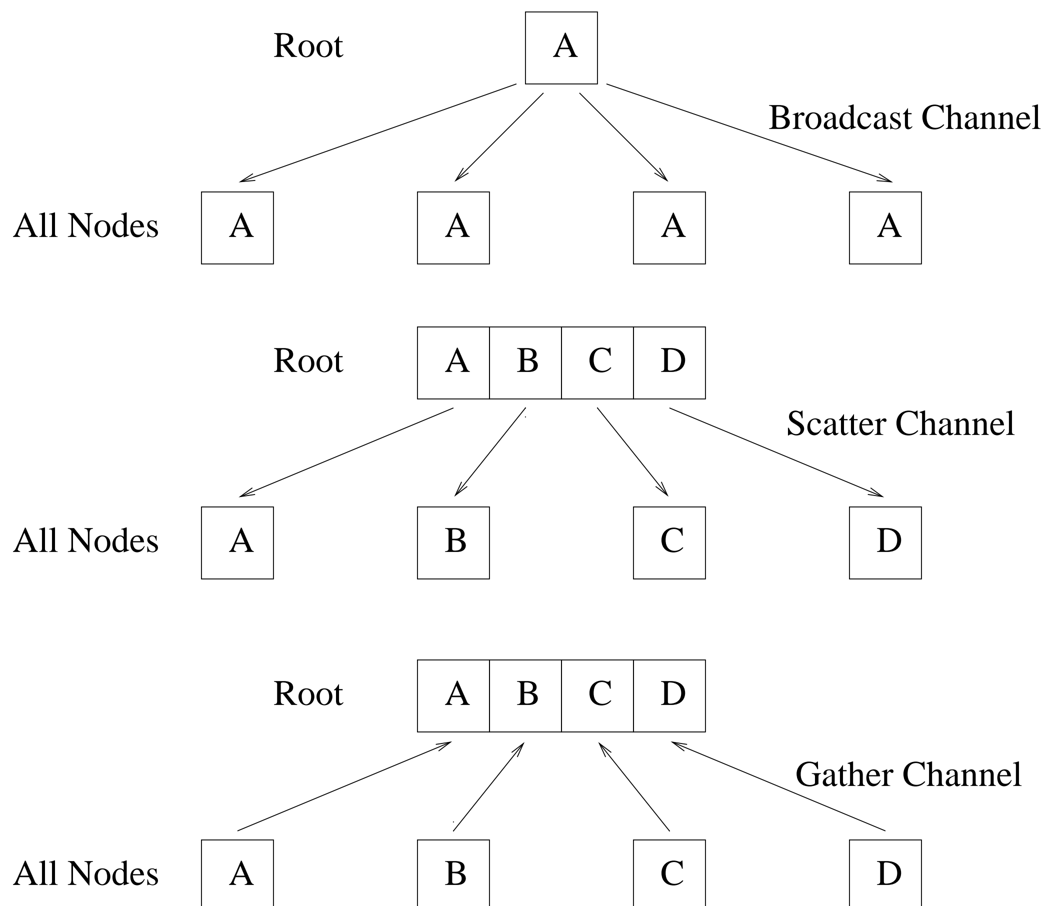

Figure 3.2 Data movement in some of the collective channels in MPI/RT.

MPI has a corresponding operation (such as `MPI_Scatter()`) for each of these channels. MPI also supports `MPI_Alltoall()` and `MPI_Gatherall()`, which can be emulated in MPI/RT 1.0 using other collective channels.

In this section, a point-to-point channel (ptchannel) is used as an example for describing the commit process and the data transfer protocol . The methods for the other channels are similar.

3.2.3 Committing a Ptchannel

Every MPI/RT object has a name (string) associated with it, which can be set by the application during creation of the object. To establish a ptchannel, both ends (sending and receiving ends) must have a Ptchannel object with the same name and the same group. During commit, MPI/RT matches the two ends of a channel based on the name and the group. This highlights the primary difference between an MPI/RT channel and MPI persistent communication (section 2.1.2), where there is no binding between both sides of the communication.

Each channel has an input buffer iterator (section 2.2.2.2), which contains buffers inserted into it. Each buffer iterator has an associated set of *allowed-buffers*. It is illegal to insert a buffer that is not a part of a buffer iterator's allowed-buffers. In the current implementation, the receiving node of a ptchannel initiates the DMA request for the data transfer. So only the receiving side of the channel needs information about the allowed-buffers on the other side. During the commit operation, the sending side sends the information

about its allowed-buffers to the receiving side. This information includes the offset of each of the buffers in the large SMB that was created.

The application might insert any of the allowed-buffers on the sending side and any of the allowed-buffers on the receiving side for a data transfer over a channel. This implies that the DX_transfer objects used for DMA transfer should be created for all the combinations of these buffer pairs. Thus, if the sending side of the channel has m allowed-buffers and the receiving side has n allowed-buffers, then mn transfer objects need to be created. The buffer iterators have a *mapped* form of each of the policies mentioned in section 2.2.2.2. A mapped policy (such as MPIRT_BUFITER_FIFO_MAPPED) for input buffer iterators on both sides of the channel implies that for any data transfer, if i^{th} buffer (in the allowed-buffer set) is used on one side of the channel then the other side also uses i^{th} buffer in its allowed-buffer set. This drastically reduces the number of DX_transfer objects that are created. There is an *ordered* form of the policies (*e.g.*, MPIRT_BUFITER_FIFO_ORDERED) where buffers are inserted into the buffer iterator in the same order in which they appear in the allowed-buffer set. Thus if the policy is *ordered*, the system knows a priori the order in which the buffers appear on both sides of the channel. Such a commitment from the application might be used to reduce the protocol overhead on some platforms. In the current implementation, *ordered* buffer iterators are treated in the same way as the *mapped* iterators when both sides have equal number of buffers in the allowed-buffer set.

MPI/RT has limited support for late binding semantics where the application may insert any buffer as long as the buffer is suitable for the channel (*i.e.*, with the same size and the same data-type). This can be achieved by specifying a *null object* for the set of allowed-buffers for the buffer iterator. In this case, the current implementation creates an internal buffer for data transfer and uses an extra copy operation.

3.2.4 MPI/RT Data Transfer Protocol

The data communication over a channel follows a 3-way protocol for communication. The three stages for a ptchannel are as follows:

1. The sending side does a remote-write (programmed I/O) to a predefined location on the receiving node indicating the buffer. This location is predefined and exchanged during the *commit* phase in MPI/RT.

2. The receiver reads the send-side buffer information from this predefined location and then resets the location. It then obtains the local buffer information and retrieves the `DX_transfer` object corresponding to this pair of local and remote buffers. A DMA transfer is initiated using this transfer object.

3. After the transfer is completed, the receive side does a remote-write to the send-side indicating that the transfer is completed. An error is indicated if the transfer failed or was not performed.

MPI/RT uses DMA for all the data sizes during the data transfer (step 2) above. This differs from MPI's approach where it uses programmed I/O for data transfers up to 32 bytes and DMA for larger transfers.

3.3 MPI Communication Subsystem

At MPI Software Technology, Inc., a commercial, portable MPI implementation has been developed where the platform independent upper layer is built over the platform dependent layer. This is further described in [5]. The platform dependent layer includes the communication layer, a major portion of which is an efficient implementation of point-to-point communication between any two nodes in the distributed system. This section briefly describes the communication layer design for Mercury with emphasis on how it differs from MPI/RT communication layer.

3.3.1 DMA Transfers

We mentioned in section 3.1.2 that on Mercury, an application needs to create shared memory buffers (SMBs) in order to initiate DMA transfers. In MPI/RT, we create one large SMB from whence we allocate the MPI/RT buffers used for data transfers. But in MPI, applications supply the buffer for data transfers. This buffer could be located anywhere in the application's memory. Since Mercury does not allow DMA transfers between arbitrary memory locations, we could use a large SMB for data exchange and copy the data to and from the application specified buffers. This scheme would be expensive as it involves an extra memory copy. To overcome this, Mercury allowed us to use an undocumented feature where we can treat the whole of a remote node's memory as an SMB. Since this is an undocumented feature, we are not sure if it incurs an extra penalty compared to a DMA transfer between regular SMBs. We expect the penalty to be insignificant in any event.

Since MPI needs to support DMA between arbitrary locations of memory, it initially creates one DMA transfer object (section 3.1.3) and modifies its parameters before each data transfer. This modification is roughly equivalent to creation of a new `DX_transfer` that is done for each pair of buffers during the MPI/RT commit phase. The overhead incurred for run time modification of the DMA object is one of the primary differences between MPI and MPI/RT DMA transfers.

3.3.2 MPI Data Transfer Protocol

In the MPI/RT protocol (section 3.2.4), the two end points of the communication use predefined memory locations for communicating the state of the channel and the buffers used for data transfer. These locations are determined for each channel during the commit phase. Because of absence of such a location for each transfer in MPI, the implementation incurs overhead of *flow control* for each each data transfer. The following section describes the flow control mechanism in MPI.

3.3.2.1 Synchronization and Flow Control in MPI

The MPI implementation allocates a fixed number of small blocks of memory, which we call *control blocks*, that are used for exchanging information on a data transfer. Each node allocates an array of control blocks for each of the other nodes in the cluster. For each data transfer, one control block is used on each side of the transfer. Since there are only a fixed number of these blocks there can be only a fixed number of outstanding transfers at

any given time. When an MPI application requests more non-blocking transfers[1], the data transfer protocol is delayed until a control block is available if all the blocks have been consumed. This flow control is not needed in MPI/RT. Flow control results in an extra remote write through programmed I/O.

Since each node needs to allocate the control blocks, these blocks are made sufficiently large so that they can hold a fixed amount of data that needs to be transferred. MPI uses a 3-way protocol similar to MPI/RT for larger data transfers and uses a shorter 2-way protocol for smaller data transfers. The 2-way protocol uses space available in the control block for transferring the data. Currently the 3-way protocol is used for messages larger than 1024 bytes.

3.3.2.2 The Transfer Protocols

The 3-way protocol is essentially the same as that in MPI/RT implementation. The sender remote-writes the information regarding the transfer and the receiver reads this information and remote-writes confirmation after completing the data transfer. The primary difference is in the amount of information that is written on the remote node's memory for the transfer. MPI needs around 28 bytes (in each direction) as opposed to 4 bytes in MPI/RT. The fields that MPI needs to write include a `tag`, a `context_id`, rank of the node, length of the data, and physical address of the buffer. Many of these fields are a direct result of the MPI semantics. As mentioned above, MPI requires another four byte remote-write for *flow*

[1] An MPI application typically starts multiple data transfers using its non-blocking communication API and later waits for these transfers to finish.

control. All the *remote-writes* mentioned above are performed through programmed I/O because it is more efficient for small data writes to the remote node's memory. Even for small messages, overhead of programmed I/O linearly increases with the size of the data. So it takes nearly four times as long to write 32 bytes than it takes for 8 bytes. The considerable difference in number of remote-writes is one of the primary reasons for higher latency in MPI. This difference is even more prominent in the case of collective communication since programmed I/O is less scalable than DMA on Mercury's RACEway network that connects all the nodes.

In the shorter 2-way protocol that is used for data sizes less than 1024, the third step is eliminated. Here, when the sender first writes the information regarding the transfer to the remote node's control block, it also transfers the actual data to the space allocated on the block. When the receiver notices that the data size is less than 1024, it just copies the data from the control block to the application's buffer instead of transferring it from the sender's node.

MPI/RT could certainly use a similar 2-way protocol for small data transfers. But the effect of this would be less dramatic in MPI/RT because, unlike in MPI, the third step in the 3-way protocol involves only four bytes of data. The manufacturer of this system evidently advises its users to minimize programmed I/O because of its issues with scalability and stability at high loads. In all of the performance results presented in this thesis, MPI uses different protocols based on the size of data whereas MPI/RT always uses the 3-way protocol.

3.4 Summary of Design Differences

This section summarizes the main differences in communication layers in MPI and MPI/RT that contribute to the differences in their performance. These are as follows:

1. MPI needs to modify the DMA transfer object before each DMA transfer.
2. MPI incurs an overhead for flow control that involves a remote-write.
3. Each side of the data transfer needs to write more data to the other side in MPI than in MPI/RT. The cost of remote-writes increases linearly with data size.

Any compliant MPI implementation has to have the key ingredients noted above, or else suffer from lack of robustness. But depending the underlying platform, such differences could have bigger or smaller impacts on performance.

CHAPTER IV

RESEARCH METHODOLOGY AND EXPERIMENTS

Our hypothesis (section 1.3) that MPI/RT can achieve better performance than MPI is verified by analyzing the performance of a set of MPI/RT and MPI applications along with latency and bandwidth measurements. This chapter outlines the various experiments that have been conducted to compare MPI/RT and MPI performance both in terms of efficiency and scalability. Scalability has been tested up to 16 nodes. It is expected that the following experiments not only illustrate performance benefits of MPI/RT over MPI, but also demonstrate the general applicability of best effort MPI/RT as an alternative to MPI for common high performance parallel applications.

4.1 Latency and Bandwidth Measurements

A simple application with two nodes is used to measure the latency and bandwidth between the two nodes. The round-trip times are measured using data transfers of various sizes. The primary aim of this experiment is to examine the overhead incurred by MPI/RT and MPI for basic point-to-point communication. The results of this experiment establish the basic characteristics of these two middleware libraries that are also reflected in other experiments that involve various forms of collective communication and data transfer patterns.

In order to obtain the results in similar environments, we use persistent communication (section 2.1.2) in MPI for data transfer.

4.2 3-D Poisson Solver

In this application, a differential equation (Poisson equation) is solved over a 3-D rectangular grid. Here, the 3-D grid is decomposed in the x, y, and z dimensions and are assigned to the participating nodes. This is a typical application that simulates data movement patterns in many scientific computing applications. The performance is measured with various sizes of the grids and with different numbers of processes. Adaptability of MPI/RT with respect to ease of programming is also examined. Since this application primarily involves point-to-point communication, we compare MPI/RT performance with MPI's persistent communication. MPI does not provide a persistent communication interface for collective communication.

4.3 RT_Cornerturn

The RT_Cornerturn benchmark, developed at MITRE [6], measures the performance of distributed matrix transposition (corner-turn). Here, an arbitrary size matrix is distributed over an arbitrary number of processes. In MITRE's MPI implementation of the benchmark, a square matrix of single precision complex data is distributed among a specified number of processes by rows. Each process performs a local transpose of its data. Then an all-to-all communication is performed using `MPI_Alltoallv()` (figure 4.1). After the

communication, each process reorganizes the data to complete the corner-turn. Now, each of the processes contains its portion of the transposed matrix. This application primarily tests the efficiency of the collective communication of the messaging layer.

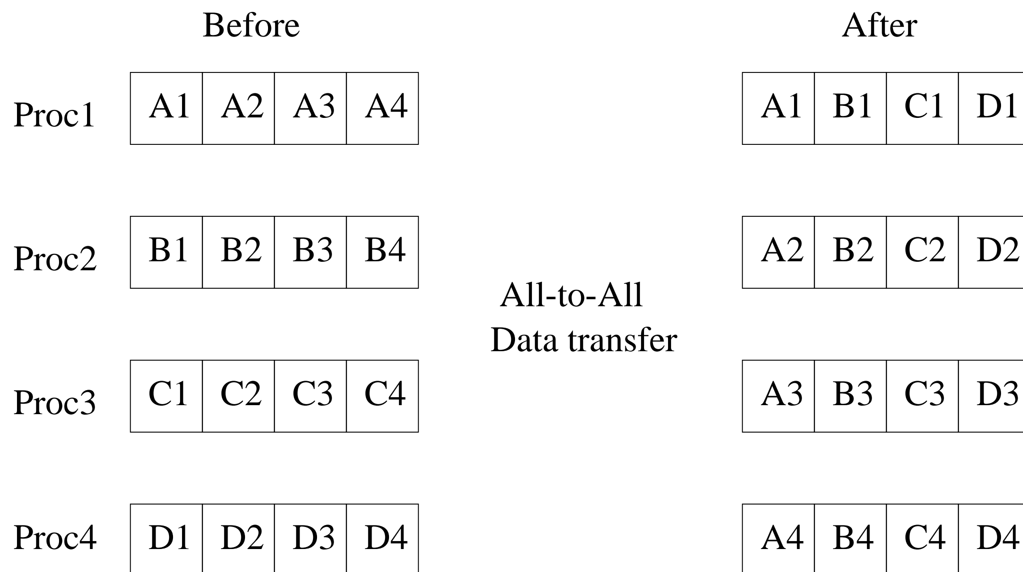

Figure 4.1 Data movement in an all-to-all communication

MPI has explicit support for all-to-all communication (figure 4.1) which is missing in MPI/RT 1.0[1]. In MPI/RT, all-to-all communication can be achieved by using N scatter or gather channels, where N is the number of processes participating in the collective communication. The performance is measured with different sizes of the square matrix and with different number of processes. We have adopted MITRE's MPI implementation of

[1] In MPI/RT 1.1, an all-to-all channel was added, but that is not exploited in this implementation.

this benchmark with minor modifications and developed an MPI/RT version for comparison.

4.4 An image processing example

An MPI/RT application, *Slab*, which performs a parallel image enhancement has been developed by SKY Computers, Inc. We have used this MPI/RT version with minor modifications and developed an MPI version to compare the performance. The outline of the application is as follows:

- The root process distributes a frame among the processes.
- Each process processes the data and then sends its local minimum and maximum to the root.
- The root process broadcasts the global minimum and global maximum values to all the processes.
- All the processes scale their parts of the image according to the global minimum and maximum.
- The root collects the image sections from all the processes.

Most of the collective channels available in MPI/RT are utilized in this application. This benchmark is expected to capture the communication and computation patterns in a typical real-time image processing applications.

CHAPTER V

RESULTS AND ANALYSIS

This chapter presents the results of various experiments described in the previous chapter. The performance benefits of MPI/RT over MPI are demonstrated with the following experiments:

1. Latency measurements for various data sizes.
2. A scientific computing application that solves Poisson equation over a 3-D rectangular grid.
3. RT_Cornerturn benchmark which transposes a square matrix distributed over multiple nodes.
4. An image processing application called *Slab*.

The following section describes the experimental set up in terms of hardware and software used. The subsequent sections describe the results of each of the experiments.

5.1 Experimental Setup

The Mercury hardware, Mercury OS (MC/OS), and MPI and MPI/RT libraries used in these experiments were provided by MPI Software Technology, Inc. The Mercury embedded multicomputer has 32 processors, where 16 of them are 300 MHz PowerPC processors and the other 16 are 400 MHz processors. Each mother board contains four processors

and the processors are connected by Mercury's proprietary internetwork called RACE-way. Each of these processors has 64 MB RAM and runs MC/OS version 5.6. In order to run the experiments using homogeneous nodes only the 400 MHz processor nodes were used. So all the applications except latency test are run with up to 16 nodes. MPIPro 1.6.3 for Mercury is used for the MPI library. The MPI/RT library used is MPI/RT-Pro 1.0 for Mercury.

The timings have been measured over a number of iterations and the average time for each iteration is represented in the graphs. All the timings shown in this chapter are in microseconds (μs). Instead of presenting one graph for each set of readings, an effort is made to provide a subset of graphs that is enough to illustrate the performance of MPI and MPI/RT. All the graphs are plotted on a logarithmic scale. Furthermore, the complete set of numerical results are provided in the appendix. The source code for MPI/RT and MPI applications developed for these experiments can be accessed at the URL provided in the appendix.

5.2 Latency

One of the primary factors that influences performance of a message passing application is the latency of message transfer between two nodes. As mentioned in section 4.1, we measure round trip latency for various sizes of data. Figure 5.1 plots these measurements and Table 5.1 shows these values for small message sizes.

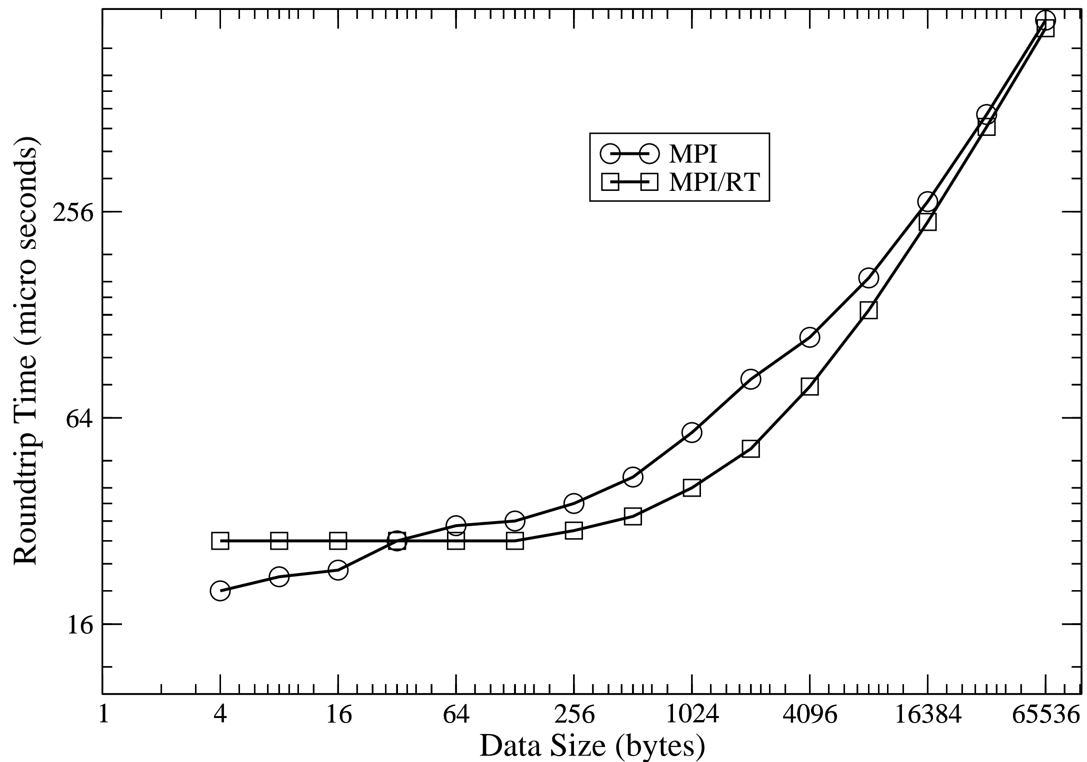

Figure 5.1 Round trip latency for MPI and MPI/RT at various data sizes

As expected, MPI has better latency for small data transfers. As described in section 3.3, MPI uses programmed I/O when the message size is less than or equal to 32 bytes and uses a shorter 2-way protocol for when the message size is less than or equal to 1024 bytes. MPI/RT always uses DMA and the 3-way protocol. Because of programmed I/O, MPI latency varies even between four and eight byte messages. On the other hand, the MPI/RT latency stays constant at 28 μs, which is essentially a measure of DMA transfer latency. The rest of the data points clearly show the benefits of MPI/RT as summarized in

Table 5.1 Round trip latency for MPI and MPI/RT

Message Size	Round Trip Latency (μs)	
	MPI	MPI/RT
4	20	28
8	22	28
16	23	28
32	28	28
64	31	28
128	32	28
256	36	30
512	43	33
1024	58	40
2048	83	52
4096	110	79

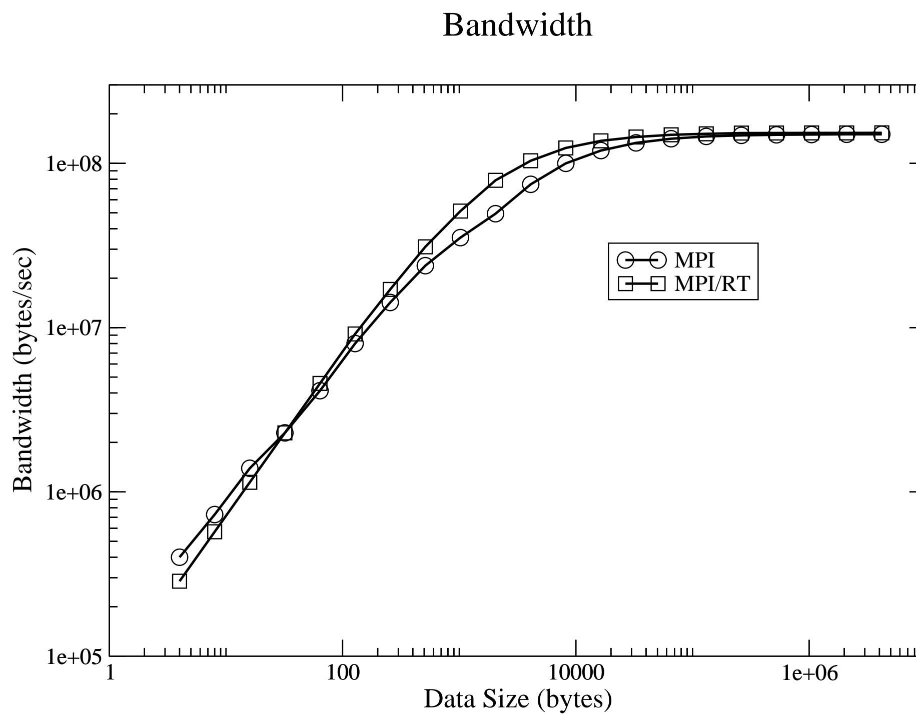

Figure 5.2 Bandwidth for MPI and MPI/RT at various data sizes

section 3.4. The primary overhead MPI incurs over MPI/RT is the cost of modifying DMA transfer parameters and more programmed I/O required for protocol synchronization. As the message size increases, the latency is dominated by the cost of actual data transfer, in which case both MPI and MPI/RT converge in performance.

In figure 5.2, the bandwidth between the nodes is plotted against the message size. The bandwidth is calculated as inverse of latency. In this setup, we are able to achieve more than 1200 Mbps one way. Even with large messages, there is a little more than 1% difference between MPI and MPI/RT. This could be because MPI needs to utilize an undocumented feature where it treats whole of the remote node's memory as an SMB and MPI/RT does not.

The results presented in this section provide strong support for our hypothesis that emphasis on early binding in MPI/RT help achieve better performance even for non-real-time applications when compared to MPI. The rest of the experiments explore how these advantages translate to better performance in more realistic message passing applications.

5.3 3-D Poisson Solver

In this application, a 3-D rectangular grid is divided into multiple rectangular cubes and distributed among all the nodes. Each node performs computation on its portion of the grid (in this case, it recalculates the value at each point based on the changed boundary values). At the end of the iteration, each node exchanges its boundary elements with its neighbors.

A node can have up to six neighbors, one in each of the three dimensions. Most of the message transfers takes place over point-to-point communication between the neighbors.

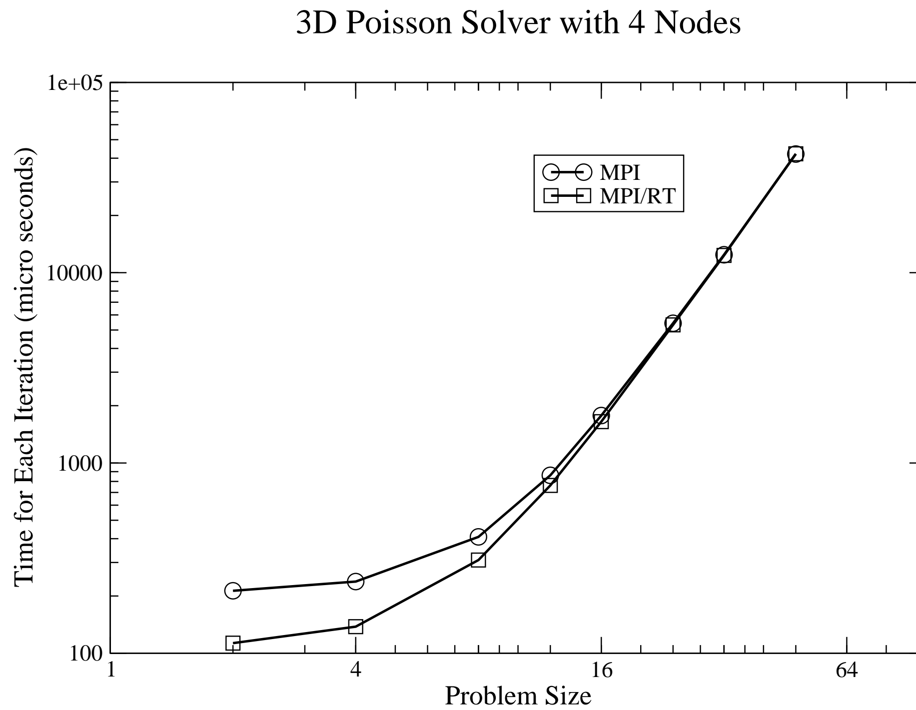

Figure 5.3 3D Poisson Solver with 4 Nodes

The data is stored in a 3-D array on each node. Since a node's neighbors could be in any one the three directions, the plane it shares with its neighbor is not necessarily contiguous in the 3-D array used for storage. While exchanging such a non-contiguous plane, we need to first *pack* the non-contiguous data into a single buffer on the send-side before the transfer and *unpack* the data on the receive-side. MPI provides a useful API where the packing and unpacking of data is completely transparent to the application.

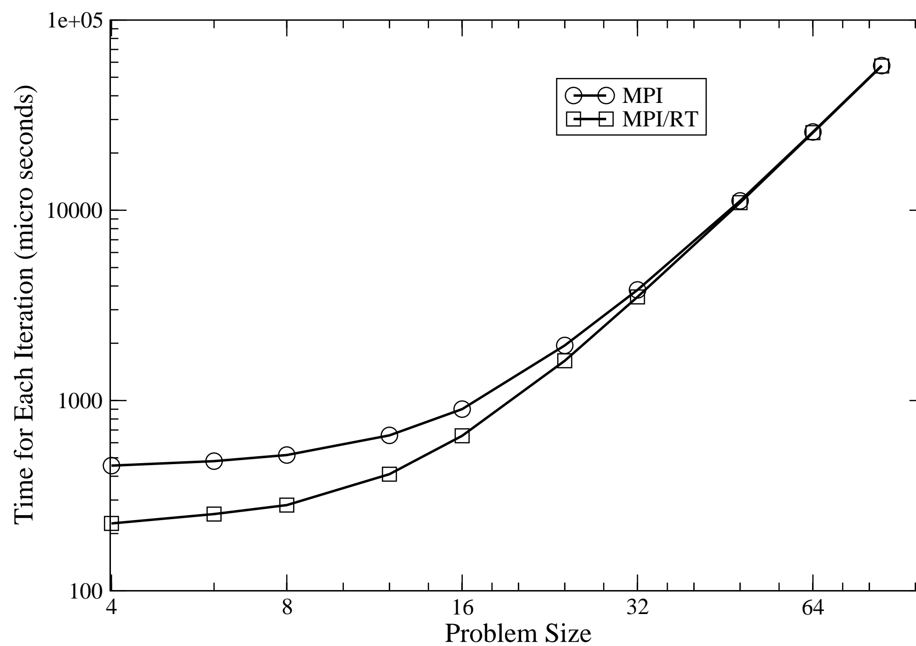

Figure 5.4 3-D Poisson Solver with 16 nodes

MPI allows applications to create new data types that can represent non-contiguous data. These data types can be used everywhere as though the data is contiguous. Since this API is missing in MPI/RT 1.0[1], we must explicitly pack and unpack the data on either side of the channel. This certainly makes MPI/RT application more complicated to write and thus more error prone. MPI/RT requirement that the application needs to commit all the resources a priori also makes application development more complicated. This is some of the cost of enforcing early binding semantics and supporting the real-time paradigm.

[1]and 1.1 versions of the standard. Improvements are suggested for MPI/RT 1.2 or 2.0 but these are not actively pursued at present.

Figures 5.3 and 5.4 show the results using four and 16 processes respectively. MPI/RT performs significantly better than MPI at smaller grid sizes. This is expected since MPI/RT has better latency variation than MPI. Table A.1 and Table A.2 provide numerical results for various configurations. The difference between MPI and MPI/RT increases with increase in the number of processes, indicating better scalability of MPI/RT. We look at scalability in more detail in the next section where performance of collective communication is more vital than it is for this solver.

5.4 RT_Conrnerturn

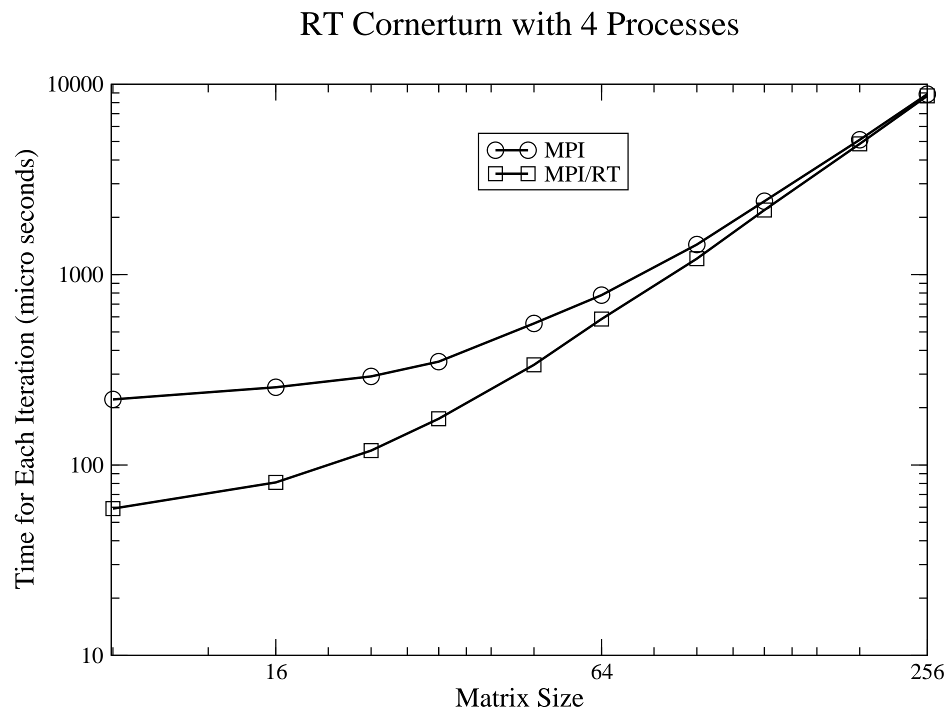

Figure 5.5 RT_Cornerturn performance with 4 processes

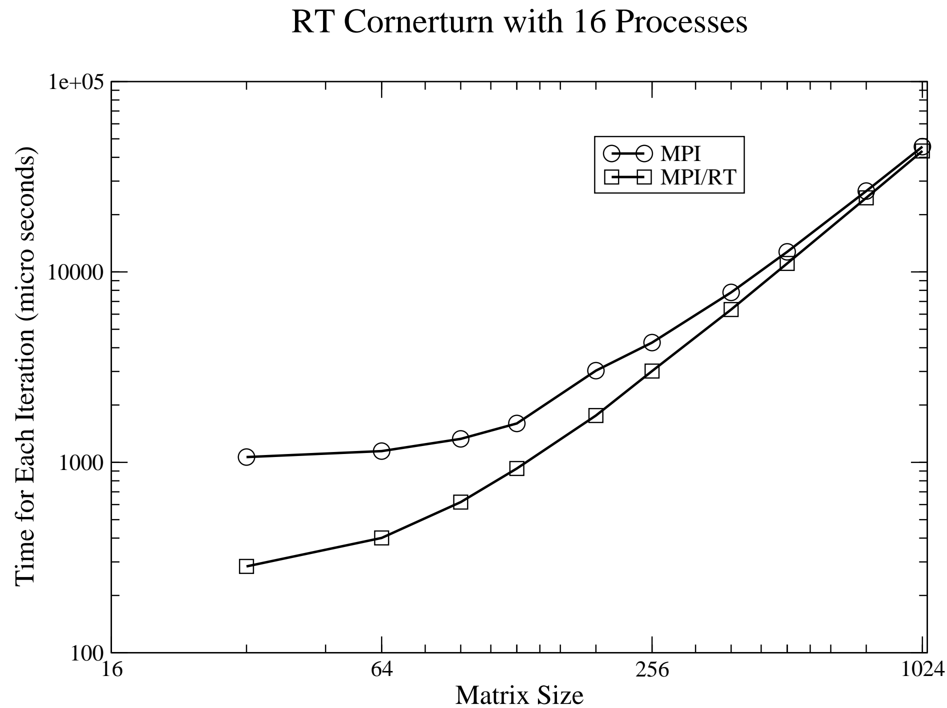

Figure 5.6 RT_Cornerturn performance with 16 processes

The RT_Cornerturn benchmark is described in section 4.3. A square matrix is distributed (by rows) among the participating nodes. The matrix is transposed and verified in each iteration. The measurements are made over 100 iterations. This benchmark stresses performance of collective communication. In MPI, all communication is performed in one invocation of `MPI_AlltoAll()` (section 4.3). The effect of all-to-all communication in MPI/RT is achieved through multiple scatter channels. The computation involves transposing the local portion of the matrix and verifying the globally transposed matrix.

The time taken for each iteration with four and 16 processes is plotted in figures 5.5 and 5.6. MPI/RT performs increasingly better with smaller sizes and more processes. The extra

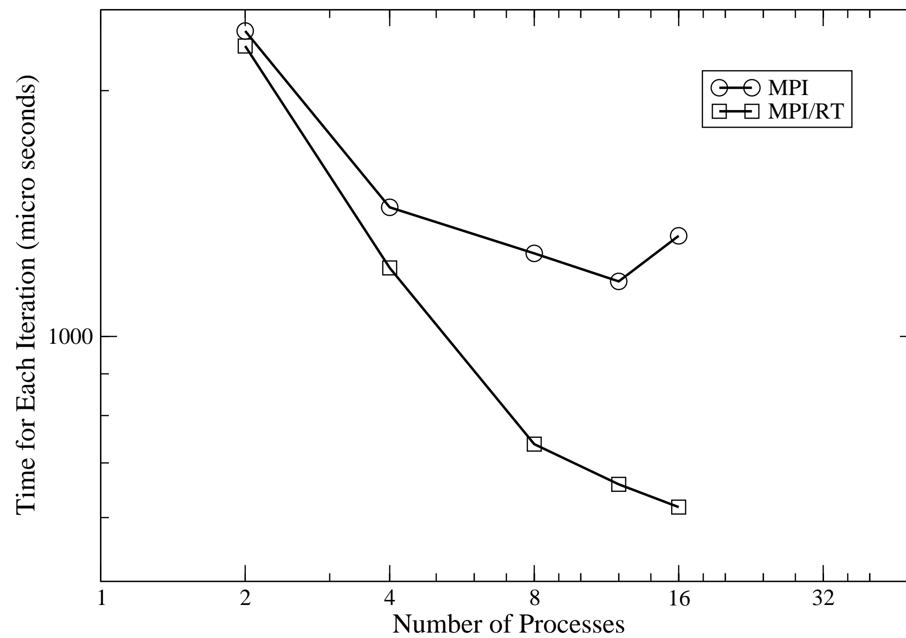

Figure 5.7 Scalability of RT_Cornerturn with matrix size 96

overhead incurred by MPI in terms of programmed I/O affects its scalability because the underlying platform scales better with DMA than with programmed I/O. Figure 5.7 plots scalability of RT_Cornerturn with a matrix size of 96. The negative scalability of MPI between 12 and 16 processes is largely due to the proportional increase in programmed I/O. Since MPI/RT involves smaller percentage of programmed I/O, it maintains better scalability. Table A.3 and Table A.4 provide the complete set of numerical results.

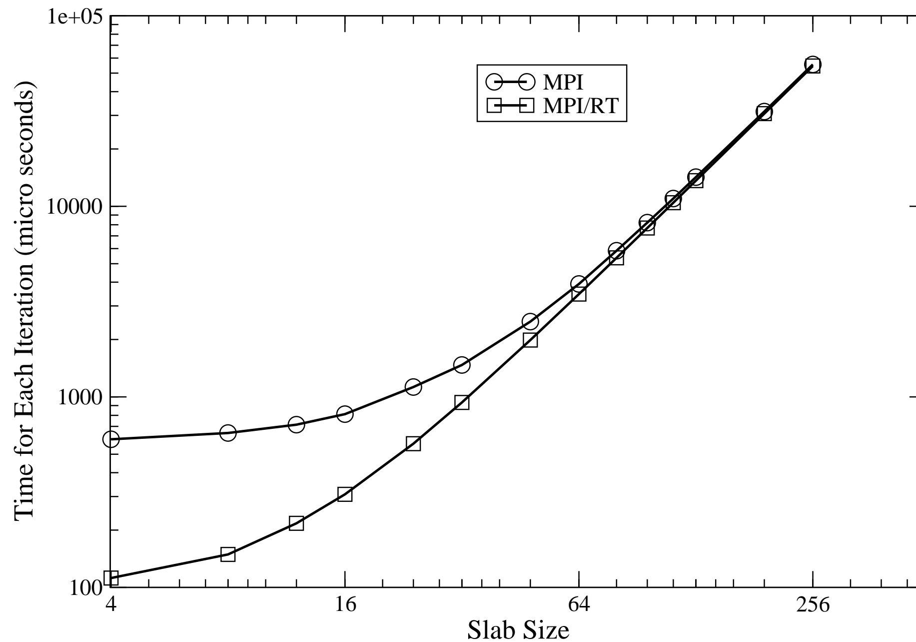

Figure 5.8 Performance of Slab with 4 processes

5.5 Slab – An Image Processing Example

This benchmark is outlined in section 4.4. *Slab* mimics an embedded image processing application where it receives a frame and the frame is distributed between all the nodes for processing. Slab uses most of the collective channels available in MPI/RT: scatter, gather, broadcast, and barrier channels. It uses an equivalent API on MPI.

Figures 5.8 and 5.9 show the results with four and 16 processes respectively. These graphs have similar characteristics as in the case of RT_Cornerturn. This is expected since in both the applications, the main emphasis is on performance of collective communica-

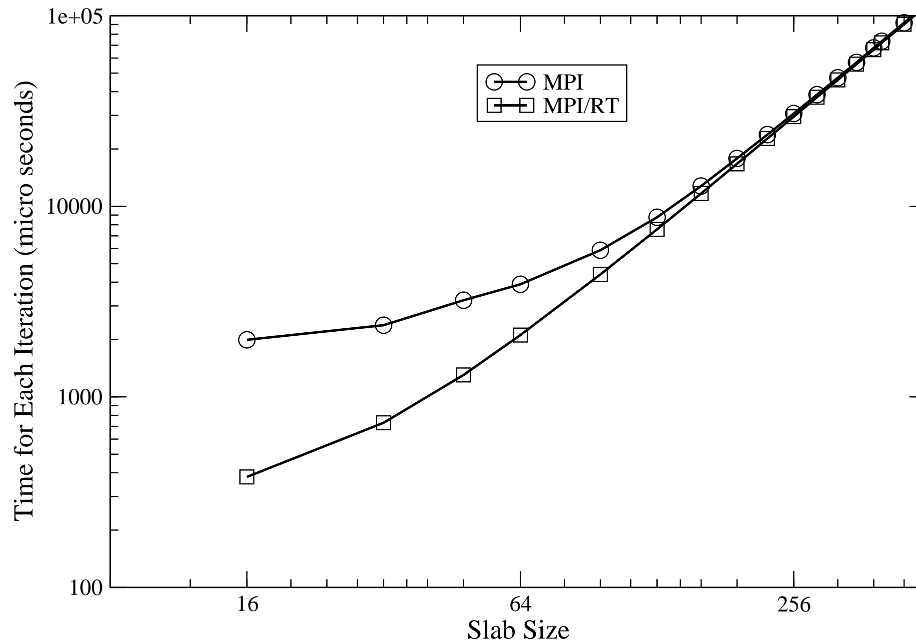

Figure 5.9 Performance of Slab with 16 processes

tion. Tables A.5, A.6, and A.7 in the appendix present the results of more configurations of this experiment.

5.6 Summary of Results

The results presented in this chapter show how syntax and semantics of MPI/RT result in better performance on Mercury platform, both in terms of latency and scalability. The results for all the experiments fall in line with the expectations. Though the round trip latency is less for MPI for small messages, MPI/RT performs better in the benchmarks at smaller problem sizes. As described in the earlier sections, this behavior is expected since

MPI/RT requires less programmed I/O. For applications involving collective communication MPI/RT shows better than 50% gains for smaller problem sizes. These results confirm both parts of the hypothesis (section 1.3).

As an application becomes more coarse-grain with increase in problem size, its performance essentially becomes bandwidth limited. In this case, the difference between MPI and MPI/RT is insignificant. The early binding semantics provide better peformance benifits for fine-grain applications on this platform.

CHAPTER VI

CONCLUSIONS

In our hypothesis, we claimed that MPI/RT with best effort QoS can be used for many parallel message passing applications and its emphasis on early binding results better performance when compared to MPI applications on the same platform. Chapter 3 describes how differences in MPI/RT and MPI semantics yield different implementations on Mercury embedded platform. It also enumerates primary advantages of MPI/RT design that contributes to its performance. The subsequent two chapters describe a set of experiments and analyze the results. These results show that MPI/RT performs considerably better in the case of fine-grain problems. For coarse-grain applications, where the performance is bandwidth limited, the difference is much less noticeable. We are more likely to encounter fine-grain problems in embedded platforms such as real-time image processing systems.

The next section presents some thoughts on MPI and MPI/RT based on the work presented in this thesis. We conclude the chapter with some suggestions for future work.

6.1 MPI vs. MPI/RT

Though this thesis clearly demonstrates MPI/RT performance advantages, we do not expect MPI/RT to replace MPI as the primary interface for message passing applications.

Two of the great virtues of MPI are its elegance and simplicity. MPI also provides a vast set of convenient API often used in parallel applications. But we do hope that work presented in this provides support for extensions to MPI that allow implementations and applications to better exploit advantages of early binding on many platforms.

While developing large scale parallel applications, software engineering advantages of programming paradigm used can outweigh performance advantages. Even though we have not developed large MPI/RT applications for this thesis, our experience shows that MPI/RT programs are usually more complicated to develop than equivalent MPI programs. This is primarily because MPI/RT is really meant for real-time applications. We expect MPI/RT will be more prevalent on platforms with low latency and high bandwidth networks than on high latency platforms like networks of workstations (NoWs). As the cost of hardware keeps on dropping, large clusters with thousands of individual commodity computers are becoming more commonplace. MPI is better suited for these platforms. The embedded platforms are usually much more restrictive and often operate in environments where adding extra nodes to increase the performance is not an easy option. MPI would be much more applicable in these environments if the standard provides extensions that utilizes precious resources better.

Initially, we had planned to compare the performance of Integer Sorting benchmark from the NAS parallel benchmarks [2]. This benchmark involves transfer of messages whose location and the size dynamically vary over time. Since MPI/RT 1.0 does not allow a communication channel to change the size of data transfered, this benchmark was not

implemented in MPI/RT. This is an example where late binding semantics provided in MPI are much more natural.

6.2 Future Work

We have measured scalability of the benchmarks up to 16 processes. Testing with larger number of nodes would provide more insight into the performance characteristics of these libraries. One of the criteria that affects a real world message passing parallel application is its ability to overlap its communication with its computation. The experiments presented in this thesis do not emphasis this overlap. It would be interesting to compare these two middleware libraries with such applications, though the ability to overlap communication with computation would depend more on the application requirements and implementation details than the semantics of these standards.

A natural extension for this thesis would be to propose and implement a set of optional extensions to MPI standard that let applications take advantage of early binding. The following aspects could be taken into account while designing such an extension:

- Persistent communication API for collective communication.
- Ability to use system buffers in communication.
- A channel abstraction that ties persistent communication API and the buffers.
- A setup phase similar to commit phase in MPI/RT.

Such extentions have already been proposed, for example, in [16]. Many of the above extensions can coexist with the current tried and tested API. An implementation can always

implement these API as wrappers over the existing API if the platform does not provide significant benefits with early binding. This would make MPI a lot more suitable for latency sensitive embedded applications.

REFERENCES

[1] M. Apte, S. Chakravarthi, , A. Pillai, A. Skjellum, and X. Zan, "Time-based Linux for Real-Time NOWs and MPI/RT," *Real-Time Systems Symposium 1999 held in Phoenix, AZ*, Dec. 1999.

[2] D. H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow, *The NAS Parallel Benchmarks 2.0*, Tech. Rep. NAS–95–020, Numerical Aerospace Simulation, NASA Ames Research Center, Dec. 1995.

[3] M. Barabanov, *A Linux-based Real-Time Operating System*, master's thesis, New Mexico Institute of Mining and Technology, June 1997.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE-Micro*, vol. 15, no. 1, Feb. 1995, pp. 29–36.

[5] R. Dimitrov, *Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations*, doctoral dissertation, Mississippi State University, May 2001.

[6] R. A. Games, *Benchmarking Methodology for Real-Time Embedded Scalable High Peroformance Computing*, Tech. Rep. MTR 96B00000010, The MITRE Corporation, 1996.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-performance, portable implementation of the MPI Message Passing Interface Standard," *Parallel Computing*, vol. 22, no. 6, Sept. 1996, pp. 789–828.

[8] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, Massachusetts, 1994.

[9] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar, "Predictable Communication Protocol Processing in Real-Time Mach," *Proceedings of 2nd Real-Time Tech., and Appl. Symposium*, June 1996.

[10] A. Mehra, A. Indiresan, and K. G. Shin, "Structuring Communication Software for Quality-of-Service Guarantees," *Proceedings of the Real-Time Systems Symposium*, Dec. 1996.

[11] Mercury Computer Systems, Inc, *Developer's Guide*, Mercury Computer Systems, Inc, Chelmsford, Massachusetts, 1997.

[12] MPI Forum, "MPI: A Message Passing Interface Standard," June 1995, `http://www.mpi-forum.org/docs/mpi-11.ps`.

[13] MPI Forum, "MPI-2: Extentions to Message Passing Interface," July 1997, `http://www.mpi-forum.org/docs/mpi-20.ps`.

[14] MPI/RT Forum, "Document for the Real-time Message Passing Interface Standard (MPI/RT-1.0) Draft Standard," June 1999, `http://www.mpirt.org`.

[15] J. P. Neelamegam, *Zero-Sided Communication: Challenges in Implementing Time-Based Channels Using The MPI/RT Spcification*, master's thesis, Mississippi State University, May 2002.

[16] A. Skjellum, "High Performance MPI: Extending the Message Passing Interface for Higher Performance and Higher Predictability," *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Las Vegas, Nevada, 1998.

[17] A. Skjellum, N. E. Doss, K. Viswanathan, A. Chowdappa, and P. V. Bangalore, "Extending the Message Passing Interface (MPI)," *Proceedings of the Scalable Parallel Libraries Conference II held in Mississippi State, MS, October, 1994*, A. Skjellum and D. S. Reese, eds. Oct. 1994, IEEE Computer Society Press.

[18] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, *MPI: The Complete Reference*, The MIT Press, Cambridge, Massachusetts, 1996.

APPENDIX

COMPREHENSIVE SET OF RESULTS

The tables provided here present more complete set of results for the experiments described in chapter 5. All the times are in microseconds. The source code for the benchmarks is available at the following URL:

`http://hpcl.cs.msstate.edu/~angadi/mpi-vs-mpirt.tgz`

Table A.1 3-D Poisson Solver Results I

Grid Size	$NP = 2$		Matix Size	$NP = 4$		Grid Size	$NP = 6$	
	MPI	MPI/RT		MPI	MPI/RT		MPI	MPI/RT
2	125	76	2	213	113	2	290	134
4	169	115	4	238	138	3	298	143
8	499	445	8	409	309	6	356	196
12	1405	1331	12	861	762	9	515	353
16	3183	3125	16	1776	1648	12	751	591
24	10351	10295	24	5425	5319	18	1856	1633
32	24595	24510	32	12430	12330	24	3884	3692
48	84426	84387	48	42026	42183	36	12039	11927
64	201065	201104	64	99956	99832	48	28286	28234
			96	339611	339422	60	54823	54790
						72	94924	95112
						84	153045	153336

Table A.2 3-D Poisson Solver Results II

Grid Size	Time taken for each iteration (μs)					
	$NP = 8$		$NP = 12$		$NP = 16$	
	MPI	MPI/RT	MPI	MPI/RT	MPI	MPI/RT
4	340	186	415	205	455	226
6	375	223	450	226	480	253
8	437	286	498	279	517	282
12	681	530	662	438	656	410
16	1139	990	990	809	901	653
24	3092	2899	2347	2040	1947	1618
32	6679	6501	4926	4661	3813	3505
48	21251	21126	14550	14393	11203	10970
64	50139	50067	34791	34732	25787	25603
84	114700	114951	76157	76272	57604	57343
96	171918	172199	115755	115127	86614	86334
128	409078	406107	278505	276657	204648	205324

Table A.3 RT_Cornernturn Results I

Matrix Size	$NP = 2$		Matix Size	$NP = 4$		Matrix Size	$NP = 8$	
	MPI	MPI/RT		MPI	MPI/RT		MPI	MPI/RT
4	108	29	8	221	59	16	484	125
8	120	40	16	256	81	32	544	172
12	138	57	24	292	119	48	626	253
16	164	83	32	349	175	64	744	375
24	257	157	48	554	336	96	1264	738
32	360	259	64	780	584	128	1757	1271
48	652	551	96	1439	1213	192	3197	2742
64	1096	994	128	2428	2184	256	5270	4821
96	2366	2268	192	5108	4859	384	11092	10497
128	4137	4038	256	8869	8681	512	19261	18788
168	6999	6933	336	15002	14775	672	32881	32840
192	9137	9045	384	19583	19409	768	42953	43032
256	16284	16193	512	34903	36097	1024	77304	78605
384	37764	38150	768	80053	80736	1536	173439	179895
512	68861	69632	1024	141191	144883			

Table A.4 RT_Cornernturn Results II

Matrix Size	$NP = 10$		Matix Size	$NP = 12$		Matrix Size	$NP = 16$	
	MPI	MPI/RT		MPI	MPI/RT		MPI	MPI/RT
20	629	161	24	788	201	32	1066	284
40	693	223	48	846	282	64	1145	401
60	801	336	72	976	442	96	1328	618
80	957	507	96	1168	659	128	1600	928
120	1719	992	144	2245	1300	192	3031	1761
160	2412	1703	192	3165	2208	256	4258	3016
240	4435	3671	288	5781	4704	384	7810	6347
320	7277	6462	384	9537	8231	512	12748	11089
480	15251	14228	576	19832	18187	768	26597	24507
640	26324	25355	768	33962	32266	1024	45512	43178
840	44784	43936	1008	57479	55672	1344	76913	74528
960	58358	57706	1152	75008	72715	1536	100202	97956
1280	104452	105143	1536	133627	130461	2048	177455	176129

Table A.5 Slab Results I

Frame Size	$NP = 2$		Frame Size	$NP = 4$		Frame Size	$NP = 6$	
	MPI	MPI/RT		MPI	MPI/RT		MPI	MPI/RT
2	337	80	4	599	112	6	885	157
4	359	95	8	646	149	12	945	220
8	423	158	12	715	217	24	1239	503
12	529	264	16	812	308	36	1756	974
16	681	411	24	1126	568	60	3180	2483
24	1115	834	32	1470	935	84	5357	4746
32	1703	1431	48	2483	1986	108	8415	7773
48	3390	3134	64	3914	3464	132	12246	11560
64	5875	5532	80	5846	5367	180	22265	21489
80	9065	8625	96	8209	7693	228	35290	34423
96	12972	12457	112	10990	10449	276	51415	50383
112	17503	16938	128	14229	13632	300	60638	59501
128	22716	22094	192	31426	30650			
192	50596	49520	256	55563	54487			
256	89800	88019						

Table A.6 Slab Results II

Frame Size	$NP = 8$		Frame Size	$NP = 10$		Frame Size	$NP = 12$	
	MPI	MPI/RT		MPI	MPI/RT		MPI	MPI/RT
8	1083	189	10	1357	247	12	1564	279
16	1193	300	20	1527	393	24	1782	494
24	1375	484	40	2282	1015	36	2161	849
32	1656	737	60	3240	2061	60	3305	1977
48	2432	1472	80	4559	3522	84	4855	3659
64	3401	2497	100	6264	5393	108	6896	5903
96	6116	5419	140	11265	10401	132	9597	8710
128	10249	9526	180	17996	17097	168	14906	13988
160	15564	14811	240	31316	30356	204	21476	20545
192	22133	21324	300	48420	47349	240	29396	28427
256	38726	37842	350	65539	83154	300	45390	44308
320	60102	59042	400	85920	84223	360	65012	63779
			500	133616	131348	420	88503	86829
						480	115156	113289
						540	145502	143219

Table A.7 Slab Results III

Frame Size	$NP = 16$		Frame Size	$NP = 16$	
	MPI	MPI/RT		MPI	MPI/RT
16	1992	380	320	47300	46077
32	2378	731	352	56948	55762
48	3217	1303	384	67861	66395
64	3900	2110	400	73503	72109
96	5883	4389	448	91827	90247
128	8766	7577	496	112226	110561
160	12779	11683	560	142667	140836
192	17831	16712	624	176782	174680
224	23782	22682	688	214579	212228
256	30701	29555	720	234856	232395
288	38564	37372			

Table A.8 Latency and Bandwidth Measurements

Message Size	Round Trip Latency (μs)		Bandwidth (bytes/sec)	
	MPI	MPI/RT	MPI	MPI/RT
4	20	28	400000	285714
8	22	28	727272	571428
16	23	28	1391304	1142856
32	28	28	2285714	2285714
64	31	28	4129032	4571428
128	32	28	8000000	9142856
256	36	30	14222222	17066666
512	43	33	23813952	31030302
1024	58	40	35310344	51200000
2048	83	52	49349396	78769230
4096	110	79	74472726	103696202
8192	164	132	99902438	124121212
16384	274	239	119591240	137104602
32768	492	453	133203252	144671080
65536	928	879	141241378	149114902
131072	1799	1733	145716508	151266012
262144	3544	3432	147936794	152764568
524288	7030	6852	149157324	153032106
1048576	14006	13696	149732400	153121494
2097152	27945	27377	150091392	153205390
4194304	55831	54737	150250004	153252972