rMPI: An MPI-Compliant Message Passing Library for Tiled Architectures

by

James Ryan Psota

B.S., Cornell University (2002)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2005

Author ....
Department of [illegible] Engineering and Computer Science
December 21, 2005

Certified by ....
Anant Agarwal
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
[illegible]. Smith
Chairman, Department Committee on Graduate Students

rMPI: An MPI-Compliant Message Passing Library for Tiled Architectures

by

James Ryan Psota

Submitted to the Department of Electrical Engineering and Computer Science
on December 21, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Next-generation microprocessors will increasingly rely on parallelism, as opposed to frequency scaling, for improvements in performance. Microprocessor designers are attaining such parallelism by placing multiple processing cores on a single piece of silicon. As the architecture of modern computer systems evolves from single monolithic cores to multiple cores, its programming models continue to evolve. Programming parallel computer systems has historically been quite challenging because the programmer must orchestrate both computation and communication. A number of different models have evolved to help the programmer with this arduous task, from standardized shared memory and message passing application programming interfaces, to automatically parallelizing compilers that attempt to achieve performance and correctness similar to that of hand-coded programs. One of the most widely used standard programming interfaces is the Message Passing Interface (MPI).

This thesis contributes rMPI, a robust, deadlock-free, high performance design and implementation of MPI for the Raw tiled architecture. rMPI's design constitutes the marriage of the MPI interface and the Raw system, allowing programmers to employ a well understood programming model to a novel high performance parallel computer. rMPI introduces robust, deadlock-free, and high-performance mechanisms to program Raw; offers an interface to Raw that is compatible with current MPI software; gives programmers already familiar with MPI an easy interface with which to program Raw; and gives programmers fine-grain control over their programs when trusting automatic parallelization tools is not desirable. Experimental evaluations show that the resulting library has relatively low overhead, scales well with increasing message sizes for a number of collective algorithms, and enables respectable speedups for real applications.

Thesis Supervisor: Anant Agarwal
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis is the result of encouragement, support, and guidance of many. I am grateful to Anant Agarwal for his mentorship and advice, and the freedom he gave me to chart my own course while implementing rMPI. Jeff Squyres, one of the key developers of LAM/MPI, served as an indispensable resource and pillar of support through the implementation of the final version of rMPI. His dedication was constant and is much appreciated. Gregory Shakhnarovich was also an incredibly helpful collaborator, contributing countless hours of brainstorming, designing, and debugging during Version II's design and implementation.

The members of the Raw group were indispensable throughout the life cycle of rMPI. Paul Johnson, Jason Miller, Hank Hoffman, Satish Ramaswamy, Patrick Griffin, Rodric Rabbah, and the rest of the Raw group all contributed an endless amount of support. Also, thanks to Alexey Radul for moral support and careful thesis reading.

Lastly, thanks to Amelia, Mom, Dad, and Julie for your love, support, and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The high performance parallel computing industry pushes and explores the frontier of computing, and is, in many cases, the crucible in which the industry's cutting edge advances form. High performance parallel computers are the holy grail of computing, often representing the most powerful computers available today. These computer systems were once used only for code breaking, nuclear weapon design, and other military operations. Over time, such systems became more commonplace in carrying out highly calculation-intensive tasks such as weather forecasting, climate modeling, financial modeling, hydrodynamics, molecular dynamics and modeling, quantum chemistry, and cryptanalysis. Furthermore, ultra-fast computers are increasingly being used for everyday product design. For instance, in early 2005, the Japanese Automobile Manufacturers Association reported that they had used their Earth Simulator supercomputer [19], currently the fourth fastest supercomputer in the world [7], to drastically increase the speed and resolution of their car crash simulations. Procter & Gamble even employed a high performance computer to investigate the airflow over its Pringles potato chips in an attempt to keep them from blowing off the assembly line! Additionally, diaper manufacturers routinely run computational fluid dynamics simulations on high performance computers in an attempt to create more effective diapers [3, 32, 11].

Research and development of high performance computers show no signs of slowing down. In addition to reasons stemming from their now widespread use, they are increasingly being viewed by nations as crucial investments for progress in national security, advanced technology, and cutting-edge science. The economic impacts of high performance computing are potentially tremendously significant. The global race towards the next milestone in high performance computing performance, breaking the petaflop[1] barrier, is intensifying. Scientists and technology executives, especially from

[1]A petaflop is a measure of computing performance equivalent to the ability to perform 1,000 trillion mathematical

the U.S., Japan, and China, believe having access to the world's fastest supercomputers is necessary to stay ahead in crucial markets like oil and gas exploration, automobile design, and manufacturing. Plus, possessing the world's fastest computer has become an international sign of status.

Such a thrust towards developing faster computers has certainly pushed and is continuing to push virtually every aspect of computing forward. The two key features that are typically used to characterize a high performance parallel computer are its architecture and programming model.

Computer architecture has seen manifest growth in the later half of the twentieth century. From burgeoning silicon resources due to advances in microelectronics, and significant innovations in computer architecture, microprocessors have seen a doubling in performance every 18 months, starting in 1965, as described by Moore's Law [4]. Over the last few years, computer architects have been using these copious silicon resources in novel ways. One of the main architectural innovations has been to improve processor performance by increasing parallelism as opposed to attempting to run processors at faster frequencies. Microprocessor designers are beginning to tile multiple processors onto a single piece of silicon in an attempt to keep wire lengths short. Designers are motivated to keep wires short because they have become the predominant factor in determining a microprocessor's speed and power use.

The other primary feature of a high performance computer is its programming model, which describes the manner in which the programmer interacts with the system. The manner in which these parallel systems, both the world's fastest ones and more commonplace smaller parallel systems, are programmed have also experienced an evolution. To take advantage of multiple processing elements, the programmer must partition the computation, allocating different portions of it to different processors so multiple threads of computation can occur simultaneously. Furthermore, the data that the computation uses as input must be physically transported to the appropriate processing elements in a coherent and timely fashion. Programmers are faced with this task of explicitly specifying parallelization and communication because doing so automatically is quite challenging. Thus, programming large parallel systems has been and continues to be difficult because the programmer must orchestrate both computation and communication simultaneously. Over time, a number of different models have evolved to help the programmer in this arduous task, from standardized shared memory and message passing application programming interfaces (APIs), to automatically parallelizing compilers that attempt to achieve performance and correctness similar to that of hand-coded programs.

One of the most widely-used programming interfaces is the Message Passing Interface (MPI)

operations a second.

[12], a standardized message passing API, available on a host of parallel computer systems. This thesis contributes rMPI, a robust, deadlock-free, high performance design and implementation of MPI for the Raw architecture [31, 30, 29] one of the new parallel computer systems on a chip. rMPI's design constitutes the marriage of the MPI interface and the Raw system, allowing programmers to employ a well understood programming model to a novel high performance parallel computer.

1.1 The Challenge of Programming Parallel Computers

Ever since the first parallel computer systems arose decades ago, the task of programming them has been quite challenging. To program a parallel system, the programmer is responsible for spatially and temporally coordinating both computation and communication. Parallel programs typically require large amounts of computation, and are therefore usually broken up into discrete pieces so that different processors can compute separate portions of the program in parallel. In order for each processor to carry out its task, it must somehow garner the necessary input data, communicate dependent data to other processors, and they must all ultimately coalesce their results. Writing programs that juggle these tasks is challenging, and for this reason, research in developing and improving new programming models for parallel computer systems is long-standing and continues to be quite active.

Parallel computer systems can be programmed in a variety of ways, but most fall into one of a few categories. In 1988, McGraw and Axelrod identified a few distinct paths for the development of application software for parallel computers [23]. The development paths include:

- Extend an existing compiler to automatically translate sequential programs into parallel programs
- Define a totally new parallel language and compiler system
- Extend an existing language with new operations that allow users to express parallelism.

Of course, each of these approaches has its advantages and disadvantages.

Research in automatically parallelizing compilers is particularly fervent, especially using imperative languages such as Fortran and C. These compilers take regular sequential programs and extract parallelism from them, orchestrate the partitioning of computation, and manage communication, all automatically. These compilers employ various techniques such as vectorization, loop analysis, and dependency graph analysis to extract both coarse-grain and fine-grain parallelism. The main advantages of this approach are that it allows previously written programs to be run directly after recompilation, and allows programmers to continue to write serial programs, which are typically much simpler to write. Examples of parallelizing compilers include Suif [16, 9], Open64 [6], and RawCC [18].

Some programming languages are innately parallel, incorporating primitives that allow programmers to implement parallel algorithms explicitly using language constructs. Some parallel languages such as Occam were developed from scratch. On the other hand, many parallel programming languages are really conventional or sequential programming languages with some parallel extensions. Examples of parallel programming languages include High Performance Fortran, Unified Parallel C, C*, Cilk, sC++, Charm++, pH, APL, NESL, CSP, and MultiLisp. Furthermore, new languages such as X10 from IBM , Chapel from Cray, and Fortress from Sun Microsystems have emerged from DARPA's HPCS project. These languages attempt to give the programmer well-defined abstractions to represent parallelism for high performance computer systems, and represent one possible trajectory of the parallel programming domain.

A more conservative approach to developing parallel programming is to extend a sequential programming language with functions that allow the programmer to create parallel processes, allow them to communicate with each other, and synchronize them. This approach is the easiest, least expensive, quickest, and most popular approach to parallel programming. New libraries can be created relatively quickly when new parallel architectures are developed, and if the library is written according to a standardized API, programmers can transfer their previous experience programming other parallel systems to new systems. These systems do have their drawbacks, of course. Because the compiler is not involved, there is no automatic error checking in the development of parallel codes. This can commonly lead to faulty programs that are tremendously difficult to debug. Nonetheless, this approach continues to be the most common and widespread amongst programmers of parallel computers. One of the most commonly used parallel programming interface standards is the Message Passing Interface (MPI), which offers parallel programming subroutines to programmers writing in sequential languages such as Fortran, C, and C++. Libraries meeting the MPI standard exist for virtually every kind of parallel computer, so programs written with MPI function calls are highly portable. It offers both blocking and non-blocking send and receive primitives, reduction operations, and collective operations such as broadcast, scatter, and gather. It also gives the user the ability to define and manipulate groups of processes, allowing sophisticated control over partitioning of computation and coordination of communication. The MPI standard was ratified in 1994, has been evolving ever since, and continues to be commonly embodied in new implementations for novel parallel computer systems.

1.2 The Raw Microprocessor

The Raw processor [31, 30, 29], developed at M.I.T., consists of sixteen identical programmable tiles that provide an interface to the gate, pin, and wiring resources of the chip through suitable high-level abstractions. Each tile contains an 8-stage in-order single-issue processing pipeline, a 32-entry register file, a 4-stage single-precision pipelined floating point unit, a 32 KB data cache, and 32 KB software-managed instruction caches. for the processor and static router, respectively. The compute processor's instruction set architecture (ISA) is similar to that of a MIPS R5000 processor. Furthermore, in an attempt to give the programmer finer-grain control over wire delay, the clock speed and size of the tiles were co-designed such that it takes a single clock cycle for a signal to travel through a small amount of logic and across one tile.

The tiles are interconnected by four 32-bit full-duplex on-chip networks, two of which are static, and two of which are dynamic. The static networks are named "static" because routes of messages are specified at compile time. On the other hand, the routes of messages sent on the dynamic networks are specified at run time. The Raw ISA gives the programmer the ability to carefully orchestrate the transfer of data values between the processing elements, as the networks are both register-mapped and tightly integrated into the bypass paths of the processor pipeline. This network interface allows the programmer a large amount of freedom and flexibility in constructing parallel programs; however, freedom and flexibility increase programmers' responsibility to write their programs in a deadlock-free and robust manner, a sometimes non-trivial task.

The Raw processor is discussed further in Chapter 2, and is extensively discussed in [31, 30, 29].

1.3 Contributions of the Thesis

This thesis contributes an MPI implementation for the Raw architecture that has several attractive features:

- **rMPI is compatible with current MPI software.** rMPI allows users to run pre-existing MPI applications on Raw without any modifications. rMPI is a nearly complete MPI implementation, implementing a large portion of the standard.

- **rMPI gives programmers already familiar with MPI an easy interface to program Raw.** As MPI is a widely-used and well-known message passing standard, many programmers are already familiar with it. Thus, they will be able to apply their skills to quickly write sophisticated programs on Raw.

- **rMPI gives programmers fine-grain control over their programs when trusting automatic parallelization tools are not adequate.** In some cases, usually for performance reasons, the programmer may desire to have full and transparent control over how communication and computation are orchestrated, making automatic parallelization methods less desirable than explicit mechanisms.

- **rMPI gives users a robust, deadlock-free, and high-performance programming model with which to program Raw.** Given the large amount of flexibility and freedom that the Raw ISA provides to the software, programmers are easily susceptible to deadlock and other plaguing issues. Even if the programmer is not familiar with MPI, rMPI provides an easy-to-learn high-level abstraction and manages the hardware's underlying complexity, obviating intense effort on the part of the programmer.

1.4 Thesis Outline

This thesis is structured as follows. Chapter 2 provides a brief overview of the Raw architecture and the MPI standard, highlighting their key features, and describing aspects particularly important to understanding rMPI in more detail. Chapter 3 develops the design, architecture, and implementation of the two rMPI prototype systems, known as Version I and Version II, while Chapter 4 describes the resulting production version of rMPI, known as Version III. Chapter 5 presents the experimental evaluation and analysis that show the final version of rMPI provides a scalable, deadlock-free, and relatively low-overhead MPI implementation for the Raw architecture. Finally, Chapter 6 concludes with lessons learned and future work.

Chapter 2

Background

In order to better understand the design and implementation presented in Chapter 3 and Chapter 4, some background information must first be understood. To that end, this chapter provides a brief overview of the Raw architecture and the MPI standard. The chapter concludes with a small MPI example program.[1]

2.1 The Raw Microprocessor

The Raw microprocessor addresses the challenge of building a general-purpose architecture that performs better on a wider variety of stream and embedded computing applications than traditional existing microprocessors. Raw exposes the underlying on-chip resources, including logic, wires, and pins, to software through a novel ISA so that it can take advantage of the resources in parallel applications. As seen in Figure 2-1, the resources on Raw are laid out in a 4-by-4 tiled arrangement. The tiles are identical and programmable, and each contains its own 8-stage in-order single-issue MIPS-style processing pipeline. Furthermore, each tile also contains a 4-stage single-precision pipelined FPU, a 32 KB data cache, two static and two dynamic networks, and 32KB and 64KB software-managed instruction caches for the processing pipeline and static router, respectively. The tiles are sized such that a signal can travel across the tile in a single clock cycle. While the Raw prototype chip contains 16 tiles, VLSI scaling will soon allow larger Raw chips to contain hundreds or even thousands of tiles.

The tiles are interconnected by four 32-bit full duplex on-chip networks, two static, and two dynamic. The static networks are for communication where the routes are specified at compile-time, while the dynamic networks are for routes that are specified at run time. Each tile is connected only

[1] Parts of this chapter borrow material from [31, 21, 25, 22].

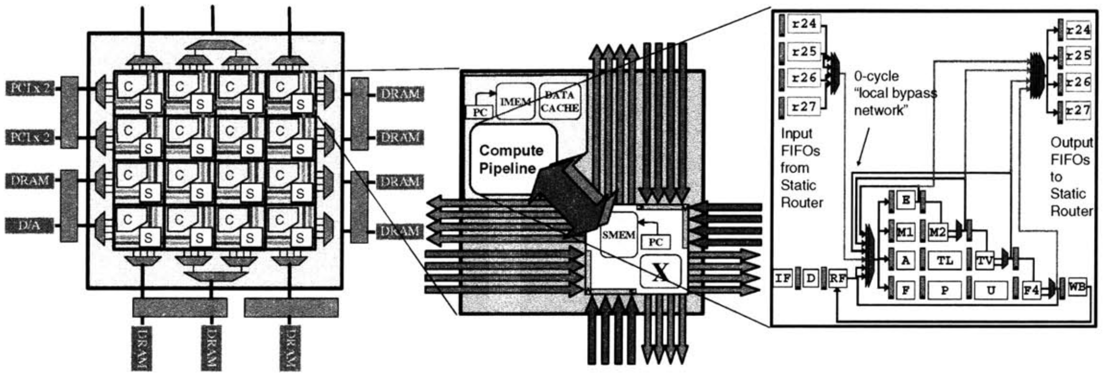


Figure 2-1: The Raw microprocessor comprises 16 tiles. Each tile has a compute processor, routers, network wires, and instruction and data memory. *Figure source: [31]. Used with permission.*

to each of its four neighbors, so data traveling from one end of the chip to the other will have to traverse multiple tiles. However, the length of the longest wire in the system is no greater than the length or width of a tile, so traversing n tiles will only take n clock cycles. This property ensures high clock speeds and the continued scalability of the architecture as more tiles are added.

Raw's on-chip network interconnect and its interface with the processing pipeline are its key innovative features. The on-chip networks are exposed to the software through the Raw ISA, giving the programmer (or compiler) the ability to directly program the wiring resources of the processor, and to carefully orchestrate the transfer of data values between the computational portions of the tiles. To go from corner to corner of the processor takes six hops, which corresponds to approximately six cycles of wire delay. Furthermore, the on-chip networks are not only register-mapped but also integrated directly into the bypass paths of the processor pipeline.

2.1.1 The Static Network

Raw's static network consists of two identical statically-scheduled networks, also known as its "scalar operand networks." These networks allow tiles to communicate with their neighbors on all four sides: north; east; south; west. Routing instructions are specified at compile-time in an instruction stream running on the switch processor, a small processor contained on each tile that handles routing. Instructions can specify to route a word in any of the four directions, or to the compute processor on the same tile. To execute an instruction, the switch processor waits until all operands are available, and then uses a crossbar to route all words in a single cycle before advancing to the next instruction. The switch processor ISA is a 64-bit VLIW form, consisting of a control flow instruction and a routing instruction. The routing instruction can specify one route for each crossbar output in a single instruction. Furthermore, because the router program memory is cached, there is no practical architectural limit on the number of simultaneous communication patterns that can be supported in a computation.

2.1.2 The Dynamic Network

Raw's two dynamic networks support cache misses, interrupts, dynamic messages, and other asynchronous events. Whereas the message destinations and sizes must be known at compile time in order to use the static network, routes and sizes of messages sent on the dynamic network can be specified at runtime. The two dynamic networks use dimension-ordered wormhole routing, and are structured identically. One network, the memory network, follows a deadlock-avoidance strategy to avoid end-point deadlock. It is only used by trusted clients such as data caches, DMA, and I/O. The second network, the general dynamic network (GDN), is used by untrusted clients, requiring the user to carefully avoid deadlock. The GDN is also supported by a deadlock recovery mechanism that is typically invoked when no forward progress has been made in the network for a certain number of cycles. However, relying on this mechanism is generally not a good idea for performance reasons, so, therefore, programs should be written to avoid deadlock if possible. rMPI does not employ the deadlock recovery mechanism; it is designed such that a well-formed MPI program will never deadlock. The remainder of this section will focus on the GDN, as it constitutes rMPI's underlying communication transport.

Details

Messages on the dynamic network are generated dynamically at runtime. Routing information is described in a message header, which encodes the length of the message, the absolute x and y coordinates of the destination tile, and the "final" route field. The final route field specifies the direction of the final route to be performed after all x-y routing is completed (*e.g.*, into processor, off chip, etc.). The length field of the GDN header is only five bits, so the maximum length of a message is 31 words (32 including the header). In the case where a message is sent to another tile, the header word is consumed by the destination tile's dynamic scheduler, so the recipient of the message never has access to it.

The programmer can access the GDN via register-mapped network ports, `$CGNI` and `$CGNO`[2], the input and output port, respectively. To send a message, the programmer uses the `IHDR` instruction to construct the message header, passing it the length and the destination tile number. Then, payload data can be sent by simply moving data to the `$CGNO` register, once per data word. By the time the payload words get into the network, the dynamic network schedulers will have already been set up by the header word. Similarly, to read data from the network, the receiving tile reads from the `$CGNI` register. If data has arrived, it can be consumed immediately. Otherwise, the receiving tile will block indefinitely.

The network guarantees that GDN messages (again, having lengths from 1 to 31 words) arrive at the destination tile in one atomic chunk. In other words, the network guarantees that multiple GDN messages arriving at the same destination tile will not be interleaved. However, if the programmer wishes to send a larger logical message, she must break it into chunks, or packets, at the sender side, and reassemble the packets into the larger logical message at the receiver side. Since the network only guarantees atomicity of individual GDN messages, packets from multiple senders may be interleaved at a destination tile. In this case, the destination tile must both reassemble and sort incoming packets. The notion of breaking larger logical messages into packets is an important one that will be seen extensively during Chapter 3 and Chapter 4, which describe the design and implementation of rMPI.

As just described, the standard way to send and receive messages over the GDN is via the `$CGNI` and `$CGNO` registers. Programmers can orchestrate communication by carefully designing a global communication pattern. For instance, one could write the appropriate code to implement the following communication pattern: "tile 0 sends 18 words to tile 8; tile 8 receives 18 words (implicitly from tile 0); tile 2 sends 1 word to tile 0; tile 0 receives 1 word (implicitly from tile 2)." With careful attention to detail, this scheme works, assuming the following constraint is met: incoming data must be consumed fast enough so that the limited in-network buffering is not overfilled. Each incoming and outgoing network port can buffer only 4 words of data. Without taking this into account, it would be easy to construct a case where deadlock ensues. This situation is described further in Section 3.2.3.

Programmers can also rely on the GDN_AVAIL interrupt to signify when new data is available on the `$CGNI` input. This interrupt fires when there is one or more data words available in the input queue of the GDN. Thus, programmers can implement programs that use an interrupt handler for all reading from `$CGNI`. In this scheme, sending messages over the GDN would occur in the same way as described above, but receiving would effectively occur in a different thread context than

[2]The register name "CGNx" is derived from "communication, general network."

the main program. Thus, data structures and other mechanisms would have to be used for communication between the data-receiving interrupt handler and the (presumably) data-consuming main thread. Such a scheme could be quite efficient, especially because receiving tiles would never block on reading from `$CGNI`, as they would only enact a read from it when data was already known to be available. However, this scheme also introduces a number of potential race conditions, and therefore requires careful timing and cautious shared data structure interaction. Chapter 3 and Chapter 4 will discuss these issues in detail. Note that one version of rMPI uses the non-interrupt-driven approach to receiving data, while the two other versions, including the resulting production version, use the interrupt-driven scheme.

2.2 The Message Passing Interface

The message passing model is one of several communication paradigms introduced in Chapter 1. Message passing is a general and powerful method of expressing parallelism through which data is explicitly transmitted from one process to another. According to this model, process A may send a message containing some of its local data values to process B, giving process B access to these values. In addition to using messages for data communication, processors also send messages to synchronize with each other. While message passing can be used to create extremely efficient parallel programs, it can be very difficult to design and develop message passing programs. It has been called the "assembly language of parallel computing" because it forces the programmer to deal with so much detail [21].

In the late 1980s, a number of parallel computer manufacturers each supplied their own proprietary message-passing library in the form of an augmentation to some ordinary sequential language (usually C or FORTRAN). Since each vendor had its own set of function calls, programs written for one parallel system were not portable to a different parallel system. After a few years of using proprietary systems, the lack of portability prompted a great deal of support for a message passing library standard for parallel computers. In 1989, the first version of PVM (Parallel Virtual Machine) was developed at Oak Ridge National Laboratory, and, after a few rewrites, became quite popular amongst parallel programmers.

Meanwhile, the Center for Research on Parallel Computing sponsored a workshop on message passing in 1992. This meeting prompted the birth of the Message Passing Interface Forum and, shortly thereafter, a preliminary draft of the Message Passing Interface (MPI) standard. Version 1.0 of the standard appeared in 1994, and has since become the most popular message-passing library standard for parallel computers. MPI is not an implementation or a language, but a standard with which implementations on different parallel computers must comply. Thus, programs written using MPI are portable: they can be moved from one parallel system to another, assuming they both offer a valid MPI implementation. Overall, such portability is a key goal of MPI, providing a virtual computing model that hides many architectural details of the underlying parallel computer.

2.2.1 What's included in MPI?

Version 1 of the MPI standard includes:

- Point-to-point communication
- Collective operations
- Process groups
- Communication domains
- Process topologies
- Environment Management and inquiry
- Profiling interface
- Bindings for Fortran and C

Point-to-point communication functions deal with the basic mechanism of MPI, the transmittal of data between a pair of processes, one side sending, the other receiving. Collective operations deal with the MPI capabilities to communicate between multiple processes simultaneously, including broadcasting, spreading, and collecting data. Process groups define the abstract collections of processes that can be used to partition the global communication domain into smaller domains to allow different groups of processors to perform independent work. These groups are created and manipulated using the abstraction of communication domains. MPI also supports mechanisms for defining logical process topologies, including grids and graphs, allowing for a standard mechanism for implementing common algorithmic concepts such as 2D grids in linear algebra. MPI also includes functions which allow for environment management and inquiry and application profiling. MPI-1 includes bindings for C and FORTRAN. Table 2.1 shows all 128 MPI-1 functions.

MPI-2, an extension to the standard born in 1995, added dynamic process management, input/output, one-sided operations, and bindings for C++. See [15] for more details on MPI-2. Overall, the complete MPI standard is massive, consisting of 280 function specifications, a myriad of datatypes and other standard-defined entities.

2.2.2 Example

While the full MPI standard includes a tremendous amount of functionality, in a large number of cases, programmers can implement a large number of programs with just six fundamental MPI functions:

- `MPI_Init` Initializes the MPI execution environment; must be called before any other MPI routine.
- `MPI_Finalize` Terminates MPI execution environment; cleans up all MPI state. Once this routine is called, no MPI routine (including `MPI_INIT`) may be called.
- `MPI_Comm_size` Determines the size of the group associated with a communicator.
- `MPI_Comm_rank` Determines the rank or id of the calling process in the given communicator.
- `MPI_Send` Performs a basic blocking send.
- `MPI_Recv` Performs a basic blocking receive.

Using these abstractions, a simple example can be constructed. In the piece of C code shown in Figure 2-2, executed by a total of two processes, process 0 and process 1, process 0 sends a message to process 1.

Again, the same code executes on both process 0 and process 1. Process 0 sends a character string using `MPI_Send()`. The first three parameters of the send call specify the data to be sent: the outgoing data is to be taken from `msg`, which consists of `strlen(msg)+1` entries, each of type `MPI_CHAR`. The fourth parameter specifies the message destination, which is process 1. The fifth parameter specifies the message tag, a numeric label used by processes to distinguish between different messages. Finally, the last parameter is a *communicator* that specifies a *communication domain* for this communication, which, among other things, serves to define a set of processes that can be contacted. Each process is labeled by a process *rank*, which are integers and are discovered by inquiry

```
int main()
{
        char msg[20];
        int myrank, tag = 42;
        MPI_Status status;

        MPI_Init(NULL, NULL);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

        if(myrank == 0)
        {
                // send message

                strcpy(msg, "Hello there");
                MPI_Send(msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
        }
        else if (myrank == 1)
        {
                // receive message
                MPI_Recv(msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

                printf("Just received message from process 0: \"%s\"\n", msg);
        }

        MPI_Finalize();
}
```

Figure 2-2: C code. Process 0 sends a message to process 1.

to a communicator. `MPI_COMM_WORLD` is the default communicator provided upon start-up that defines an initial communication domain for all processes that participate in computation.

The receiving process specifies that the incoming data is to be placed in `msg` and that it has a maximum size of 20 entries, of type `MPI_CHAR`. Finally, the variable `status` gives information on the source and tag of the message, as well as how many elements were actually received.

Using the six functions employed in this simple example program, a multitude of message passing programs can be written. The rest of the MPI functions serve to abstract away lower level details, offering high-level primitives such as broadcast, scatter/gather, and reduction. These higher-level operations are discussed more thoroughly throughout the remainder of this thesis.

MPI_ABORT	MPI_ADDRESS	MPI_ALLGATHER
MPI_ALLGATHERV	MPI_ALLREDUCE	MPI_ALLTOALL
MPI_ALLTOALLV	MPI_ATTR_DELETE	MPI_ATTR_GET
MPI_ATTR_PUT	MPI_BARRIER	MPI_BCAST
MPI_BSEND	MPI_BSEND_INIT	MPI_BUFFER_ATTACH
MPI_BUFFER_DETACH	MPI_CANCEL	MPI_CARTDIM_GET
MPI_CART_COORDS	MPI_CART_CREATE	MPI_CART_GET
MPI_CART_MAP	MPI_CART_RANK	MPI_CART_SHIFT
MPI_CART_SUB	MPI_COMM_COMPARE	MPI_COMM_CREATE
MPI_COMM_DUP	MPI_COMM_FREE	MPI_COMM_GROUP
MPI_COMM_RANK	MPI_COMM_REMOTE_GROUP	MPI_COMM_REMOTE_SIZE
MPI_COMM_SIZE	MPI_COMM_SPLIT	MPI_COMM_TEST_INTER
MPI_DIMS_CREATE	MPI_ERRHANDLER_CREATE	MPI_ERRHANDLER_FREE
MPI_ERRHANDLER_GET	MPI_ERRHANDLER_SET	MPI_ERROR_CLASS
MPI_ERROR_STRING	MPI_FINALIZE	MPI_GATHER
MPI_GATHERV	MPI_GET_COUNT	MPI_GET_ELEMENTS
MPI_GET_PROCESSOR_NAME	MPI_GRAPHDIMS_GET	MPI_GRAPH_CREATE
MPI_GRAPH_GET	MPI_GRAPH_MAP	MPI_GRAPH_NEIGHBORS
MPI_GRAPH_NEIGHBORS_COUNT	MPI_GROUP_COMPARE	MPI_GROUP_DIFFERENCE
MPI_GROUP_EXCL	MPI_GROUP_FREE	MPI_GROUP_INCL
MPI_GROUP_INTERSECTION	MPI_GROUP_RANGE_EXCL	MPI_GROUP_RANGE_INCL
MPI_GROUP_RANK	MPI_GROUP_SIZE	MPI_GROUP_TRANSLATE_RANKS
MPI_GROUP_UNION	MPI_IBSEND	MPI_INIT
MPI_INITIALIZED	MPI_INTERCOMM_CREATE	MPI_INTERCOMM_MERGE
MPI_IPROBE	MPI_IRECV	MPI_IRSEND
MPI_ISEND	MPI_ISSEND	MPI_KEYVAL_CREATE
MPI_KEYVAL_FREE	MPI_OP_CREATE	MPI_OP_FREE
MPI_PACK	MPI_PACK_SIZE	MPI_PCONTROL
MPI_PROBE	MPI_RECV	MPI_RECV_INIT
MPI_REDUCE	MPI_REDUCE_SCATTER	MPI_REQUEST_FREE
MPI_RSEND	MPI_RSEND_INIT	MPI_SCAN
MPI_SCATTER	MPI_SCATTERV	MPI_SEND
MPI_SENDRECV	MPI_SENDRECV_REPLACE	MPI_SEND_INIT
MPI_SSEND	MPI_SSEND_INIT	MPI_START
MPI_STARTALL	MPI_TEST	MPI_TESTALL
MPI_TESTANY	MPI_TESTSOME	MPI_TEST_CANCELLED
MPI_TOPO_TEST	MPI_TYPE_COMMIT	MPI_TYPE_CONTIGUOUS
MPI_TYPE_EXTENT	MPI_TYPE_FREE	MPI_TYPE_HINDEXED
MPI_TYPE_HVECTOR	MPI_TYPE_INDEXED	MPI_TYPE_LB
MPI_TYPE_SIZE	MPI_TYPE_STRUCT	MPI_TYPE_UB
MPI_TYPE_VECTOR	MPI_UNPACK	MPI_WAIT
MPI_WAITALL	MPI_WAITANY	MPI_WAITSOME

Table 2.1: Complete list of MPI-1 functions

Chapter 3

Design and Implementation of Prototype Systems

Using the motivations and observations from Chapter 1 and incorporating the features of the Raw architecture, this Chapter develops the design, architecture, and implementation of rMPI's two prototype systems. While rMPI's application programming interface (API) is fixed by the MPI standard, its implementation must strive to make the most of Raw's resources in order to produce a robust, MPI-standards-conforming, and high performance system. This chapter describes how these requirements ultimately transitioned to the precursor to the resulting design of rMPI, and show how practical issues dealing with the underlying Raw architecture influenced its implementation.

During the design and implementation process, a number of goals were established to provide direction and focus. The resulting design would be considered a success if the following objectives were reached:

- **MPI-Standard Compliant.**
- **Complete MPI Implementation.**
- **Scalable.**
- **Deadlock-Free.**
- **Optimized for the Raw Architecture.**
- **High Performance.**
- **Low Overhead.**

- **Extensible.**

rMPI has undergone a number of complete re-implementations throughout its development. The ultimate implementation is the most robust, efficient, and standards-compliant version of the three versions that have existed since the rMPI project began. However, this evolution provides insights into the final design and therefore deserves some attention. Version I is the initial rendering that implemented, from scratch, a minimal set of the MPI standard in a somewhat inefficient manner; Version II implemented a larger set of the MPI standard from scratch in a more efficient manner; Version III implements a large portion of the MPI standard in a robust and efficient manner, and is thoroughly evaluated in Chapter 5.

This chapter focuses on Versions I and II of rMPI, while Chapter 4 focuses on the third and final version of rMPI. Sections 3.2, 3.3 highlight the two prototype versions. Version I and Version II are discussed in some detail, and qualitatively analyzed at the ends of their respective sections. Version III is discussed in much more detail in the next chapter, as it is the culmination of rMPI. Version III is quantitatively analyzed in Chapter 5. Throughout this chapter, relevant details of the MPI specification are discussed when appropriate throughout discussion of different aspects of the design and implementation. Finally, the explanations of Versions I, II, III build on each other, and are intended to be read sequentially.

3.1 rMPI System Overview

rMPI is a runtime system that allows programmers to write parallel programs for the Raw architecture using the MPI API [13]. It abstracts away the potentially complicated mechanics of a process sending data to one or more other processes using the high-level interface defined by the MPI standard specification. To use rMPI, programmers simply use ld to link in the rMPI library into C programs which gives them access to the MPI routines. In fact, most off-the-shelf MPI programs can be run on Raw using rMPI[1].

The system architecture can be seen in Figure 3-1. At the top-most layer exists the user program level, where the programmer's MPI program runs. This level contains a C program, which contains calls to the MPI system (and potentially other libraries, of course). The MPI layer contains the implementation of the high-level MPI API. Such API functionality falls into a number of different classes,

[1]Save for some caveats and limitations of rMPI relative to the full MPI standard. These limitations are described in Section 4.5.

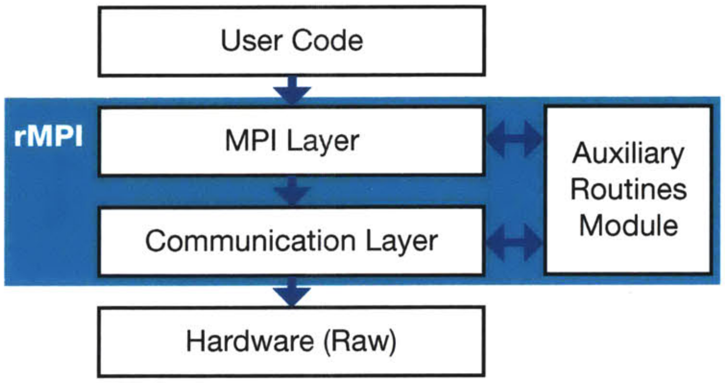


Figure 3-1: rMPI System Architecture

which are briefly described and exemplified below. Note that [13] contains the full definitions of these functions, which are left out here for clarity.

Point-to-Point Communication

These functions deal with the basic mechanism of MPI, the transmittal of data between a pair of processes, one side sending, the other receiving. Examples of point-to-point functions include `MPI_Send`, `MPI_Recv`, and `MPI_Probe`.

User-Defined Datatypes and Packing

These functions deal with sending noncontiguous memory locations in a single communication action by generalizing the predefined MPI datatypes into ones that can be defined by the user. Examples of functions that deal with datatypes include `MPI_Type_contiguous`, `MPI_Type_vector`, and `MPI_Type_create_hindexed`.

Collective Communications

These functions deal with the MPI capabilities to communicate between multiple processes simultaneously, including broadcasting, spreading, and collecting data. Examples of collective communication functions include `MPI_Gather`, `MPI_Alltoall`, and `MPI_Reduce`.

Communicators

These functions deal with the MPI abstraction of communication domains, which allow one to subdivide the processes into groups as well as protect against message collision in different libraries. Examples of functions dealing the the management of communicators include `MPI_Group_intersection`, `MPI_Comm_create`, and `MPI_Comm_split`.

The MPI Layer interacts with the Auxiliary Routines Module, which, as shown in Figure 3-1, is really more of a component of rMPI as opposed to a proper layer. The Auxiliary Routines Module provides useful abstractions to the MPI Layer (and the Communication Layer, as will be explained below) by implementing high-performance data structures such as hash tables, linked lists, queues, and arrays; efficient math operations, such as prime number generation and primality testing; run-time reporting, such as help messages that assist the end user in determining the source of run-time errors by reporting call stacks and printing useful error messages.

The MPI Layer does not interact directly with the hardware to send and receive data over the network. Rather, the Communication Layer is responsible for moving all data from one process to another. Furthermore, the Communication Layer performs the arduous task of the bookkeeping and data structure management, which is required to ensure messages are received in the proper order, deadlock is prevented, and buffer space is managed robustly and efficiently. The Communication Layer is discussed in more detail in Section 4.3.3.

Finally, the Hardware Layer contains the actual hardware, including the Raw microprocessor and its associated infrastructure (Raw motherboard, FPGAs, DRAM, host interface, etc.). Note that this layer is sometimes simulated in software using btl, the Raw cycle-accurate simulator, but in most cases the simulator yields the same results. More details about Raw can be found in the literature [31, 30, 29]. rMPI uses the general dynamic network (GDN), described in Chapter 2, as messages in MPI are inherently dynamic. Other interfaces, such as C-Flow, are available for programming Raw's static network, but these interfaces must have communication patterns that are known at compile time.

3.1.1 Nomenclature

Before delving into the details of the design and implementation of Versions I, II, and III, some terminology must first be defined.

word A 32-bit value. This quantity is the unit of transmission on the GDN.

sequence of words A juxtaposed sequence of 32-bit values.

packet A message sent on the GDN consisting of 1 to 31 words. Packets are guaranteed by the network to be received in the order in which they were sent relative to a sender-receiver pair, and to be received atomically (*i.e.,* data from two packets destined for the same tile will not be interleaved).

MPI message A collection of packets that make up a higher-level logical message. This term is used generally when referring to an MPI message, which can be of arbitrary length. The underlying assumption is that messages are broken into packets when they are greater than 31 words. Sometime referred to simply as a "message."

packet header A collection of words prepended to packets used to label and identify them. Packet headers contain information that allows receivers of packets to associate them with a larger logical MPI message. Sometimes referred to simply as "header."

payload The data portion of a packet.

process An abstract computing entity, often a processor and associated memory space. In rMPI, a tile constitutes a process.

function prototypes MPI defines function prototype bindings for three languages: Fortran, C, and C++. rMPI only implements bindings for the C language.

3.2 Version I

The primary goal of the first version of rMPI was to support what the MPI literature [25, 14] often refers to as the "six function MPI". This minuscule subset of the MPI library is powerful enough to allow the user to write many meaningful MPI programs. These six functions, with brief descriptions, are as follows:

- `MPI_Init` Initializes the MPI execution environment; must be called before any other MPI routine.
- `MPI_Finalize` Terminates MPI execution environment; cleans up all MPI state. Once this routine is called, no MPI routine (including `MPI_INIT`) may be called.
- `MPI_Comm_size` Determines the size of the group associated with a communicator.

- `MPI_Comm_rank` Determines the rank or id of the calling process in the given communicator.
- `MPI_Send` Performs a basic blocking send.
- `MPI_Recv` Performs a basic blocking receive.

Given the goal of implementing such a small subset of the API, Version I was developed from scratch using C and assembly code (as opposed to attempting to port a complete MPI implementation).

3.2.1 Sending

As described in Section 2.1.2, messages with a length over 31 words must be broken into packets. The potential for packet interleaving on the GDN necessitates the identification of each packet by the receiver. In Version I there exist two packet types, the ***first packet*** packet type and the ***non-first packet*** packet type. The first packet sent in any MPI message (sent using rMPI) contains a ***full MPI header***[2]. Figure 3-2(a) shows the format of the first packet sent in any MPI message, including the full MPI header (5 words) and the payload portion of the packet. The full MPI header contains all of the information that the receiver needs in order to correctly identify and process the incoming message. Note that the first packet can contain at most 26 payload words. Therefore, if the entire MPI message contains $N \leqslant 26$ payload words, the first (and only) packet would contain $5 + N$ words total. The full MPI header contains the following fields:

src The rank of the sending process.

tag The user-defined tag of the MPI message.

size The length of the entire message's payload, in words.

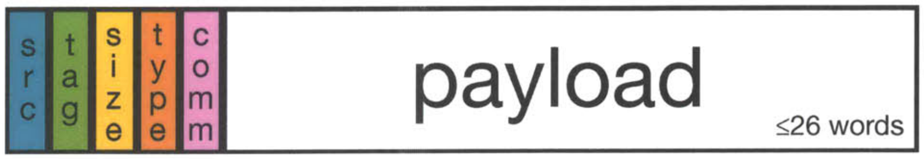
type The MPI_Datatype

comm The communication context id[3].

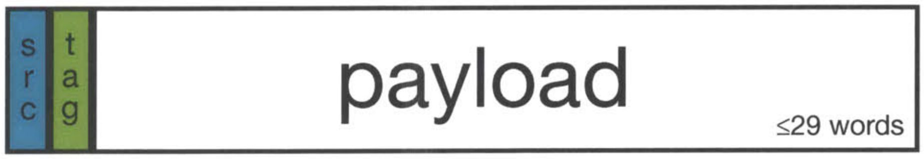
If the message payload is greater than 26 words, the sending process sends additional packets until the entire message has been transmitted. Because of the bookkeeping done by the receiver, it

[2]The packetization and packet header scheme described here refers to the specific implementation of rMPI. The MPI standard itself does not specify such implementation details.

[3]Version I only supports the `MPI_COMM_WORLD` communication context.



(a) First packet



(b) Non-first packet

Figure 3-2: Packet formats in Version I implementation

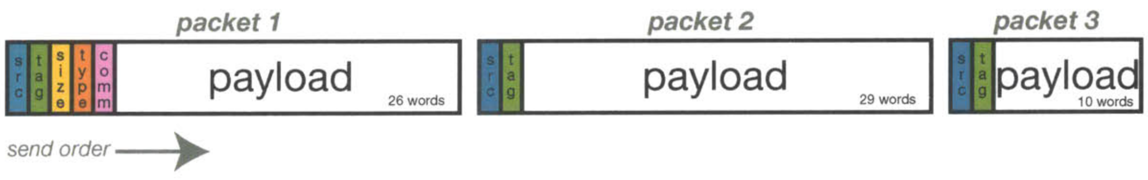


Figure 3-3: Example of a 70 word payload message (Version I implementation)

turns out that the sender only has to send a smaller header on later packets such that the receiver can still appropriately associate them with the corresponding first packet of the same message. This optimization saves processing time on the sender and receiver, and also reduces network congestion. Such "non-first" packets have a so-called *MPI mini-header*, as seen in Figure 3-2(b). The fields in this header, `src` and `tag`, have the same meaning as that in the full MPI header, and are purposely in the same position as the full MPI header for reasons dealing with the receive protocol, described in Section 3.2.2. Note that the last packet in any multi-packet MPI message can have $N \leqslant 29$ words in its payload.

Thus, the sending algorithm is quite straightforward: the sending process progresses through the data buffer to be sent, breaking it up into packets and inserting headers at the appropriate packet boundaries. For example, a message of length 70 will be broken up into three packets, with payload lengths of 26, 29, and 10 words in each of the three packets, respectively. This example is illustrated in Figure 3-3.

3.2.2 Receiving

The sending algorithm and protocol established in Section 3.2.1 must be coupled with an appropriate receiving algorithm, which is the topic of this section. Not surprisingly, it turns out the receiver's job is far more complex than that of the sender. While the GDN guarantees that GDN messages (or "packets" in the larger context of an MPI message) up to a length of 31 words will arrive at the receiver in one contiguous chunk, it guarantees nothing about how packets from multiple senders are interleaved amongst each other. Therefore, the receiving process must keep track of all partially received and unexpected messages from all sending processes. Furthermore, the receiver must distinguish between messages of different tags from the same sender, and make sure that multiple messages from a given sender with a given tag are buffered in first-in-first-out order. All of these factors contribute to a fairly complex data structure on the receiver side where bookkeeping information and the incoming data itself is stored. A high-level diagram of this structure, called the Outstanding Messages Structure (OMS), is seen in Figure 3-4.

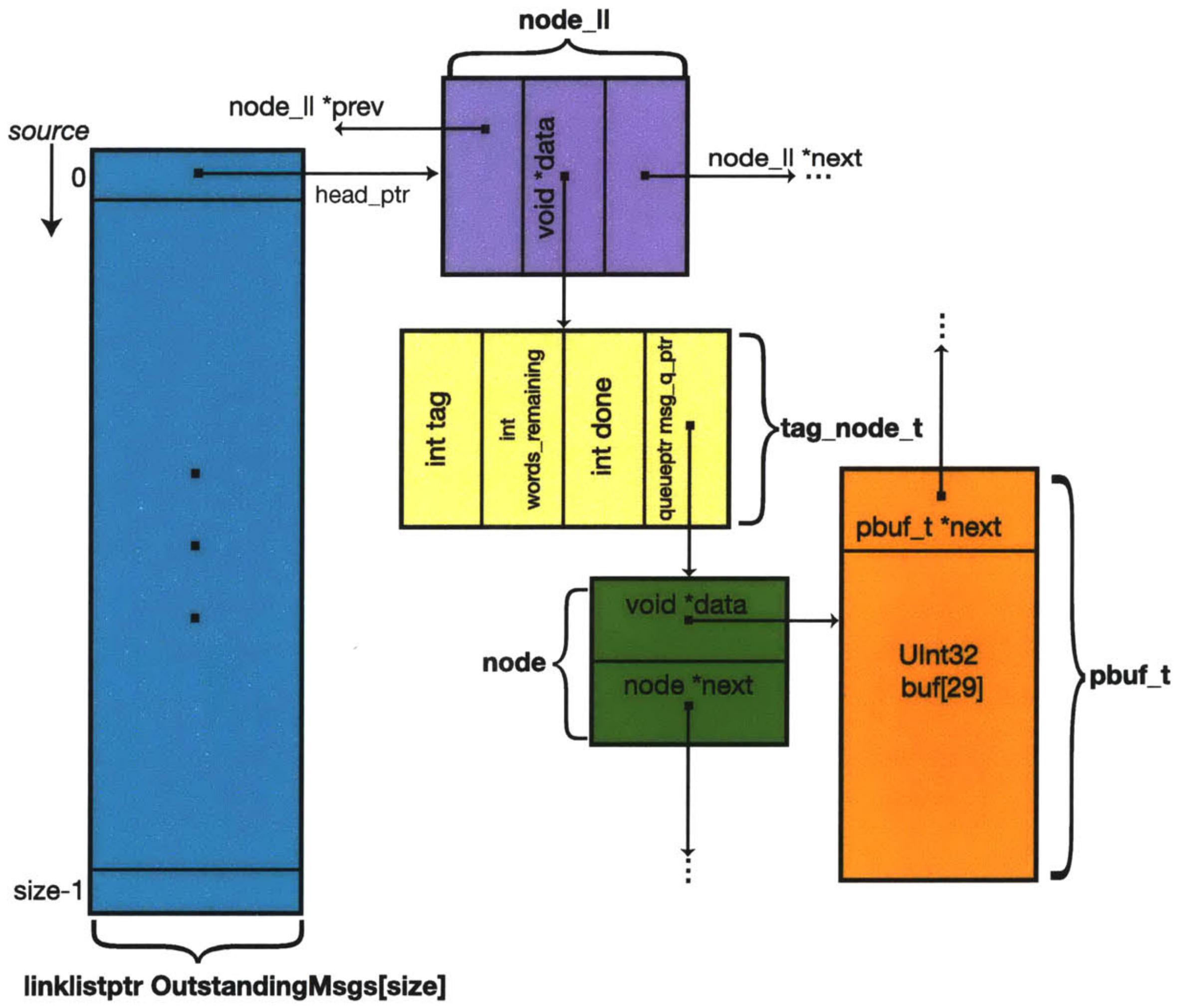


Figure 3-4: Outstanding Messages Structure, which keeps track of incoming and completed messages from all sources. The labels are consistent with those in the source code (Version I implementation).

When a process makes a call to MPI_Recv, it specifies a number of parameters describing the message it desires. The function prototype for MPI_Recv is as follows:

```
int MPI_Recv(void *buf,                /* receivers message buffer */
             int count,                /* number of elements to receive */
             MPI_Datatype datatype,    /* datatype of elements */
             int source,               /* source process */
             int tag,                  /* tag of desired message */
             MPI_Comm comm,            /* communication context */
             MPI_Status *status)       /* status report container */
```

Upon a call to this function, the receiving process attempts to receive data from the incoming GDN network port. If there is no data available, the receiving process simply blocks until data arrives. If and when data is available on the GDN input port, the receiving process reads everything it can from the network. As it is reading, it expects the packets to be formatted according to the protocol described above, with MPI headers prepended to the beginning of each message. The first packet that arrives for a given MPI message contains a full MPI header, which contains all of the information specified in the function prototype of MPI_Recv. The receiver can use the information in this initial packet to fully identify the message. However, the receiver first only reads the first and second words of the incoming packet (src and tag) and looks up in the OMS to determine if this packet is for a new message or a packet that is part of a message that has already been partially received. In the first case, the lookup into the OMS will return NULL. The receiver then knows the next three words are part of the packet header, and not payload, and receives them.

Before discussing how the OMS is initialized, a brief overview of the OMS structure itself is necessary. At the left of Figure 3-4, there exists an array named OutstandingMsgs that has the same length as the number of processes participating in the MPI program (size - 1). This structure is dynamically initialized when MPI_Init is called. Furthermore, this structure is indexed by the source of the sender of an incoming message (the src field of the header). The element of the array, OutstandingMsgs[src], contains a pointer to a doubly-linked list, whose contents is a pointer to a tag_node_t structure. A tag_node_t structure contains bookkeeping information for a given message, including its tag and the number of words that must still be received until the message is complete[4]. Each tag_node_t also contains a pointer to a queue of pointers to pbuf_t

[4]There is an important, subtle reason why we need both a words_remaining field and a done field, which has to do with receiving and copying the final packet of any MPI message.

structures, the buffers where the actual message payload resides. As an optimization, instead of making a call to malloc each time a new message arrives, the receiver dynamically allocates a pool of pbuf_t buffers, and reuses the buffers over and over. This optimization, which is a common technique that computer scientists employ in performance-critical applications, significantly reduces the number of system calls while the system is receiving data from the network (clearly a performance-critical section of the system).

When the receiving process has successfully garnered the entire MPI header, it initializes the OMS for that given source and tag. It then begins storing the incoming payload, one word at a time, into one of the pbuf_ts associated with the message. While the receiver is reading payload data, it keeps track of how many words it expects that packet to contain, and how many have already been received, so it can tell when the entire packet has been completely received. At that point, the receiver is ready to read the next packet from the network, and the entire process repeats. Note that if the receiver reads the first two words of a new packet and notices that there is already an incomplete message that has already been partially received from the same source and containing the same tag, it knows that the third word that is read off the network is data payload, as any non-first packet contains only a two-word MPI mini-header.

Upon a call to MPI_Recv, the receiver enters into the above procedure until the message that it desires has been fully received. Of course, packets not meeting the parameters described by the parameters passed to MPI_Recv could arrive, in which case the receiver simply places them in the appropriate place in the OMS.

Finally, this design also supports the MPI wildcards, MPI_ANY_SOURCE and MPI_ANY_TAG, which, as the names imply, allow the receiver to receive messages from any source or any tag (or both), by specifying these values in the src and tag fields of the call to MPI_Recv. This functionality was achieved by incorporating special checks for these cases, and performing the appropriate lookups. For example, if MPI_ANY_TAG was specified for a particular source src, the OMS would be indexed by src and, if one existed, the first message encountered would be taken. If no message was available, the receiver would process the first message that arrived from the source src. If MPI_ANY_SOURCE was specified for a specific tag, the receiver would linearly search the structure for a message from any source with the specified tag. Lastly, if both wildcards are supplied to MPI_Recv, the OMS is searched for any complete message. If no complete message exists, the receiver waits to accept the first complete message that eventually becomes available.

3.2.3 Analysis of Version I

Version I of rMPI overall successfully accomplished its goals of the six function MPI, outlined in Section 3.2. While the implementations of these functions do not fully comply with the MPI Standard, some fairly substantial MPI programs can still be run using Version I. However, further analysis reveals some flaws in its design. First of all, it's clear from Figure 3-4 that lookups and insertions into the OMS can be quite complex. The array index by the source of the incoming message into the `OutstandingMsgs` array is clearly $O(1)$, but the traversal of the linked list is $O(N)$, where N is the number of outstanding messages from a given source. Furthermore, for each of the $O(N)$ lookups that must occur, the `tag_node_t` must be inspected to determine the message state. Thus, if N is large, such as is the case when a given process sends hundreds of messages to one other process that does not receive them in the same order that they were sent, both insertions and deletions from the OMS can be quite expensive. On the other hand, if the user program is written such that the receives are done in about the same order as the sends, and the sends and receives are, for the most part, synchronized with one another, there should be few `tag_node_t` objects at any given time on the OMS, and therefore performance should not suffer much. Nonetheless, this structure is $O(N)$ in lookup speed.

A number of tests were run to verify correct operation. For instance, to ensure that packet interleaving for messages with a wide range of sizes was properly managed, many-to-one and all-to-all tests were run. Additionally, experiments to test wildcards, buffering for very large message sizes, and other basic functionality tests all proved successful.

It turns out, however, that there is a much more serious problem with the Version I implementation: in some cases, it can deadlock. The deadlock case arises because of buffer space limitations in the GDN network. Recall from Section 2.1.2 that at most 8 words can exist in the network between two adjacent tiles. If two adjacent tiles are sending messages of length greater than 8 words to each other, and they start sending their data at the same time, the situation shown in Figure 3-5 can arise.

Of course, there are various other cases that can fail. For instance, if the situation above arises but the messages are much longer, even if the sends do not occur at the same time, the buffers can become full before either tile starts receiving. This case also extends to non-adjacent tiles. The further the tiles are apart, the more buffer space exists in the network. Clearly, for each of the $\binom{16}{2} = 120$ pairs of tiles, there exists some maximum message length where this deadlock situation occurs. Because of this buffer space limitation, various other deadlock situations can arise when more than one pair

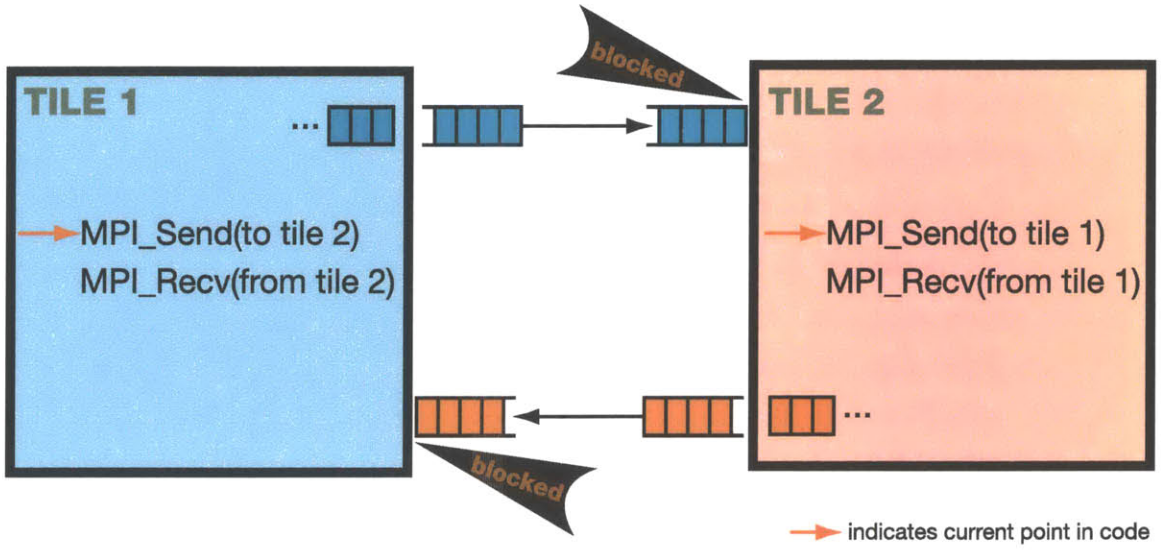


Figure 3-5: Trivial deadlock situation that can occur in Version I due to limited buffer space in GDN. Two adjacent tiles initiate an `MPI_Send` to each other of messages more than 8 words at the same exact time, and buffer space becomes full before they are able to complete their respective sends.

of tiles are sending and receiving messages. Luckily, the programmer can mitigate this problem by enabling the GDN watchdog timer. The watchdog timer monitors deadlock situations on the GDN by counting the number of consecutive cycles when no forward progress has been made. At some point a specified maximum number of "dead cycles" is reached, and an interrupt fires and program execution jumps to a handler which drains the GDN into the appropriate tile's local memory in order to allow forward progress to occur. While this facility will allow the Version I implementation to run deadlock free (assuming the user program is well-behaved), details regarding the watchdog timer show that it significantly reduces performance because of the extra overhead. Ultimately, an MPI implementation that was inherently deadlock free regardless of buffer space limitations in the architecture is desirable. This goal is approached in the Version II and Version III implementations of rMPI.

Overall, it is apparent that Version I is not quite the robust and complete MPI implementation that most MPI programmers would desire. Its internal data structures such as the OMS will most likely perform adequately for many applications, but may not do well in others. Finally, in some cases, this implementation can deadlock. These concerns lead to the design and implementation of Version II, which is discussed in the sequel.

3.3 Version II

While Version I met its original goals of implementing the six-function MPI library, it has a number of drawbacks and limitations, as discussed above. Version II was redesigned from the ground up, and sought to improve and expand upon Version I in the following ways:

Interrupt-driven design. Version II's design is fundamentally different than Version I's, as Version II's receives are completely interrupt-driven. This design removes the inherent deadlock potential exhibited in Version I, and, in many cases, allows for a higher performance design because the interrupt handler mechanisms in Raw are relatively low overhead.

O(1) **message lookups.** Version II's Outstanding Message Structure (OMS) was re-designed to enable significantly faster insertions, lookups, and removals, relative to Version I.

Optimizations and Improved Architecture. Version II adds some optimizations and a cleaner architecture relative to Version I's implementation.

Extended functionality. Version II adds more functionality from the MPI specification.

While Version II was inspired from Version I, about 80% of it is completely new. The following sections briefly describe the main differences and improvements in Version II's design.

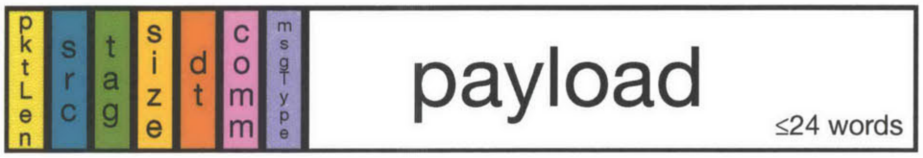
3.3.1 Sending

The packetization scheme for Version II is similar to that of Version I; there exist two packet types, the *first packet type* and the *non-first packet* type. However, to support the additional functionality and interrupt-driven design, additional fields were added to the header. As in Version I, the first packet contains a *full MPI header*, which allow the receiver to fully identify a new incoming message. Figure 3-6(a) shows the format of the first packet sent in any MPI message in Version II's implementation. It contains 7 words of header and at most 24 words of payload. The full MPI header contains the following fields:

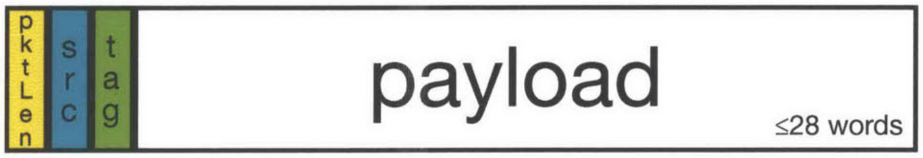
pktLen The length of this GDN message, in words.

src The rank of the sending process.

tag The tag of the MPI message.



(a) First packet



(b) Non-first packet

Figure 3-6: Packet formats in Version II implementation

size The length of the entire message's payload, in words.

type The MPI_Datatype

comm The communication context id[5].

msgType The type of message (*e.g.*, point-to-point, broadcast, gather, etc.).

The five middle fields, `src`, `tag`, `size`, `type`, and `comm`, all have the same meanings as those in Version I. However, the first and last words in this header are new. The `pktLen` field contains the number of words in the current packet; this value is used by the interrupt handler on the receiver to determine packet boundaries. This process is described more below. The `msgType` field contains an enumerated type which describes the type of message the sender is sending. This field was added to allow the receiver to differentiate between the numerous different classes of communication that can occur between two processes, including normal point-to-point sends and receives between two processes, and collective operations such as `MPI_Gather` and `MPI_Bcast`. For example, consider the case when process 1 initiates an `MPI_Send` to process 2, and also initiates an `MPI_Bcast`. The receiver will internally initiate two receives, one to receive the message from the point-to-point send, and one to receive the message from the broadcast, and must have some way of differentiating between the two incoming messages. The `msgType` fields employed in Version II addresses this need, and include support for point-to-point operations, broadcast, scatter, gather, and reduce.

[5]Version II only supports the `MPI_COMM_WORLD` communication context.

If the message payload is greater than 24 words, the sending process sends additional packets until the entire message is transmitted. As in Version I, the headers on packets other than the first don't have to contain a full MPI header; they need to contain just enough information for the receiver to appropriately associate them with the corresponding first packet of the same message, thus minimizing the amount of overhead for sending a message. The MPI mini-header can be seen in Figure 3-6(b). The fields in this header are `pktLen`, `src`, and `tag`, all of which have the same meaning and position as the fields in the first packet. As shown in the figure, non-first packets can have up to 28 words in their payload.

The sending algorithm for Version II is essentially the same as that for Version I, where the sending process progresses through the data buffer to be sent, breaking it up into packets and inserting headers at the appropriate packet boundaries. The receiving algorithm used in Version II, however, is quite different, and is described next.

3.3.2 Receiving

As in Version I, the sending algorithm and protocol established above must be coupled with an appropriate receiving algorithm. Version II's receiving algorithms and data structures are similar in spirit to Version I's, but contain a few key differences. First, and probably most fundamentally different from Version I, is that data is received via the GDN_AVAIL interrupt, which fires when data arrives on the receiver's GDN input port. When GDN_AVAIL fires, control flow jumps to an interrupt handler which drains the GDN completely, processing packets as they arrive. In fact, all of the message management functionality that was in the `MPI_Recv` code in Version I was moved into the interrupt handler in Version II. The interrupt handler is somewhat complex due to the fact that packets from different messages can interleave, and also because individual packets may be received over the course of many calls to the interrupt handler. The packet interleaving issue was solved in Version I by labeling each packet (the first of a message or otherwise) with a small header; this design decision was preserved for Version II. However, the issue of individual packets being received over multiple invocations of the interrupt handler presented another challenge.

To illustrate the issue of individual packets being interleaved and being received over multiple calls to the interrupt handler, consider Figure 3-7. This figure shows two messages, labeled msg1 and msg2, destined for Tile 2. In this example, assume that each message contains one packet, and therefore each contains a full MPI header. The partitions show how the packets are received by Tile 2. First, the initial chunk of 6 words of msg1 arrives, the interrupt fires, and all 6 words are read from

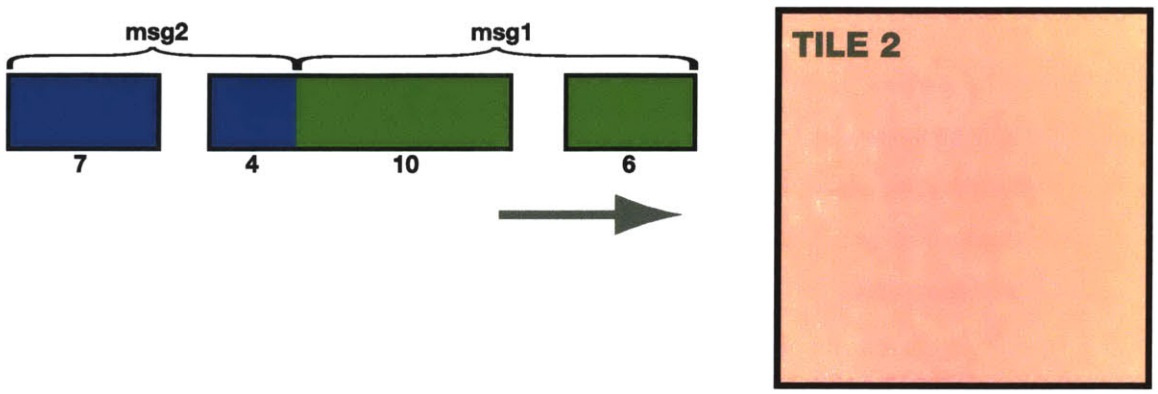
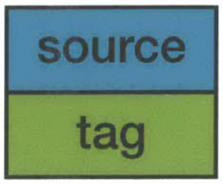


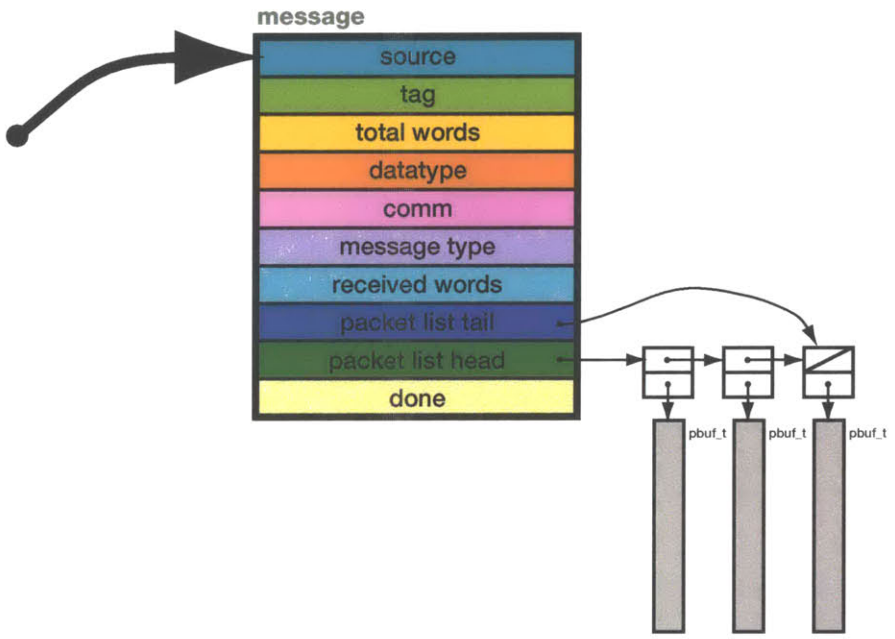
Figure 3-7: Two messages being received by Tile 2 over the course of three separate interrupt handler invocations. The first chunk that arrives contains 6 words, and is fully part of msg1. The second chunk contains 14 words, 10 of which belong to msg1, and 4 of which to msg2. Finally, the third chunk contains 7 words, all belonging to msg2.

the GDN. Upon reading the appropriate status register and determining that no more data words are available on the GDN input port, the interrupt handler then returns. At some later time, the second chunk of data arrives on the GDN input of Tile 2, and the GDN_AVAIL interrupt handler fires. This next chunk contains 14 words, but only the first 10 are from msg1, while the last 4 are from msg2. The interrupt handler uses the pktLen field of the header of each packet, which it originally read when the first chunk containing the header arrived, to appropriately differentiate between msg1 and msg2. When msg1 has been received in full, the interrupt handler temporarily stops receiving data and places msg1 into the Outstanding Message Structure (the organization of the OMS is described below). The handler then returns to receiving data from the network. After the first four words of msg2 are received, the handler notices that there are no more data available, and returns. Finally, the GDN_AVAIL interrupt fires for the third time, and the remainder of msg2 is read and stored in the OMS.

Recall that the OMS keeps track of messages that are unexpected, incomplete, and those that are complete but not yet explicitly requested by the receiver process. Version II re-implements the OMS in an attempt to achieve greater performance and a more straightforward architecture. To achieve constant time insertions, lookups, and deletions, a decision was made to employ a hash table in Version II's design. The keys in the OMS hash table consists of the source and tag of the message,



(a) Key



(b) Value (pointer to message object)

Figure 3-8: Version II Outstanding Message Structure hash table key and value definition

as seen in Figure 3-8(a). In an attempt to minimize conflicts, the hashing algorithm used in the Java 1.4 [20] hash table implementation was augmented and used for the OMS hash function. The change from the normal hash function is the section where the source field of the key is shifted left and ORed with the tag field of the key, allowing both the source and tag to be incorporated. This function was chosen because it is collision free and computationally inexpensive. Figure 3-9 shows the implementation of this function.

The values in the OMS hash table are pointers to message objects, which are depicted in Figure 3-8(b). Message objects contain all information that exists in the full MPI header. They additionally contain the following fields: pointers to the head and tail of a linked list of packet objects (the location where data is stored); the number of words received so far; and a flag that specifies if the message

```
unsigned int hash(struct hashtable *h, key *k)
{
        unsigned int i = (k->src << 21) | (k->tag);

        i += ~(i << 9);
        i ^=  ((i >> 14) | (i << 18));
        i +=  (i << 4);
        i ^=  ((i >> 10) | (i << 22));

        return i;
}
```

Figure 3-9: Hash function used in Version II Outstanding Message Structure (OMS). Algorithm derived from Java 1.4 hash table implementation.

has been entirely received thus far. The linked list of packets is similar to the structure employed in Version I, as it uses a pool of reusable buffers (`pbuf_ts`) to minimize memory allocation and deallocation overhead.

When a new packet arrives in its entirety, the interrupt handler makes a lookup in the OMS to see if this packet is the start of a new message, or the continuation of a message that has already been partially received. The receiver determines that a new packet is part of an already-initialized message if the following conditions hold true:

- The source and tag from the new packet must match the source and tag of a message in the hash table. This is determined simply by looking up a message in the hash table with the incoming source and tag to determine if a message object already exists with those same fields.
- The message type field of the message returned from the OMS must match the message type field of the packet. This will differentiate between, say, a broadcast from tile 1 to many other tiles, with a tag of 42, and a point-to-point send from tile 1 to the receiving tile, with a tag of 42. These situations must be identified, necessitating the msgType field.
- The `done` field of the message returned from the OMS must be FALSE. If this field is TRUE, the incoming packet is a packet for a new message with the exact same source, tag, and potentially message type fields as the one that matched from the OMS lookup.

If any of these conditions does not hold true, the just-arrived packet is the start of a new message. This scheme relies on the fact that the network guarantees that messages that are sent will arrive in complete form at the receiver, no portion of which will be duplicated.

Using this logic, there is no need for a flag in the header that specifies if the packet is the start of a new message, or the continuation of an uncompleted one. Thus, if the fourth word of a packet, when interpreted as an integer, contains the value 184, either there are 184 words in the message (in the first packet case), or the first payload word contains the value 184 (in the non-first packet case). The interpretation is entirely dependent upon context. This certainly complicates the design somewhat, but has the advantage of keeping the number of header words to a minimum.

If the above conditions hold, and a just-arrived packet is determined to be a part of an uncompleted message, the new packet is appended to the packet list for the associated message. On the other hand, if any of the above conditions fail, the packet is the start of a new message. When the first packet of a new message fully arrives, the receiver creates and initializes a new message object. The initial values for the source, tag, total words, datatype, communication context id, and message type fields are simply copied directly from the message header to the corresponding fields of the message object. Furthermore, the received words field is initialized to the number of words received in the first packet (as that many words have just been received), the packet list pointers are initialized to point to the first packet (the place at which the data from the first packet was just placed), and the done field is initialized to FALSE. Finally, the initialized message is inserted into the OMS with a new key object, created with the source and tag of the new message.

It should be clarified that when a new packet arrives, a new `pbuf_t` packet buffer is allocated, and data is read directly into it. The sorting of this packet into the appropriate location in the OMS is done after the entire packet arrives. As the data is read directly into the packet buffer, there is no need to copy the data; the packet is simply enqueued onto the packet list of the appropriate message object.

In order for Version II to support wildcards, some additional data structures were added. The reason for this is clear: the OMS's hash table is indexed by a specific source and tag, but wildcards are, by definition, a *non-specific* source and/or tag. The generic hash table implementation that was used for the OMS has no way of extracting an arbitrary value given a source wildcard, tag wildcard, or both source and tag wildcards. Thus, the OMS was extended from one hash table to three hash tables: the "canonical" hash table, accessed when given a fully-specified source and tag; the "source wildcard" hash table, accessed when given a source wildcard and a fully-specified tag; the "tag wildcard" hash table, accessed when given a fully-specified source and a wildcard tag. The wildcard hash tables are only keyed upon the fully-specified value; the source wildcard hash table is keyed upon the given tag, and the tag wildcard hash table is keyed upon the given source. Upon receiving the start of a message, all three hash tables are initialized to point to the new message, using appropriate keys. The following example will help to clarify how the three hash tables are initialized. Given a new message with a source of 7 and a tag of 314, the canonical hash table would get a pointer to the new message keyed on a $\langle 7, 314 \rangle$ object, the source wildcard hash table would get the same pointer keyed the tag value of 314, and the tag wildcard hash table would get the same pointer keyed the source value of 7. This process is depicted in Figure 3-10.

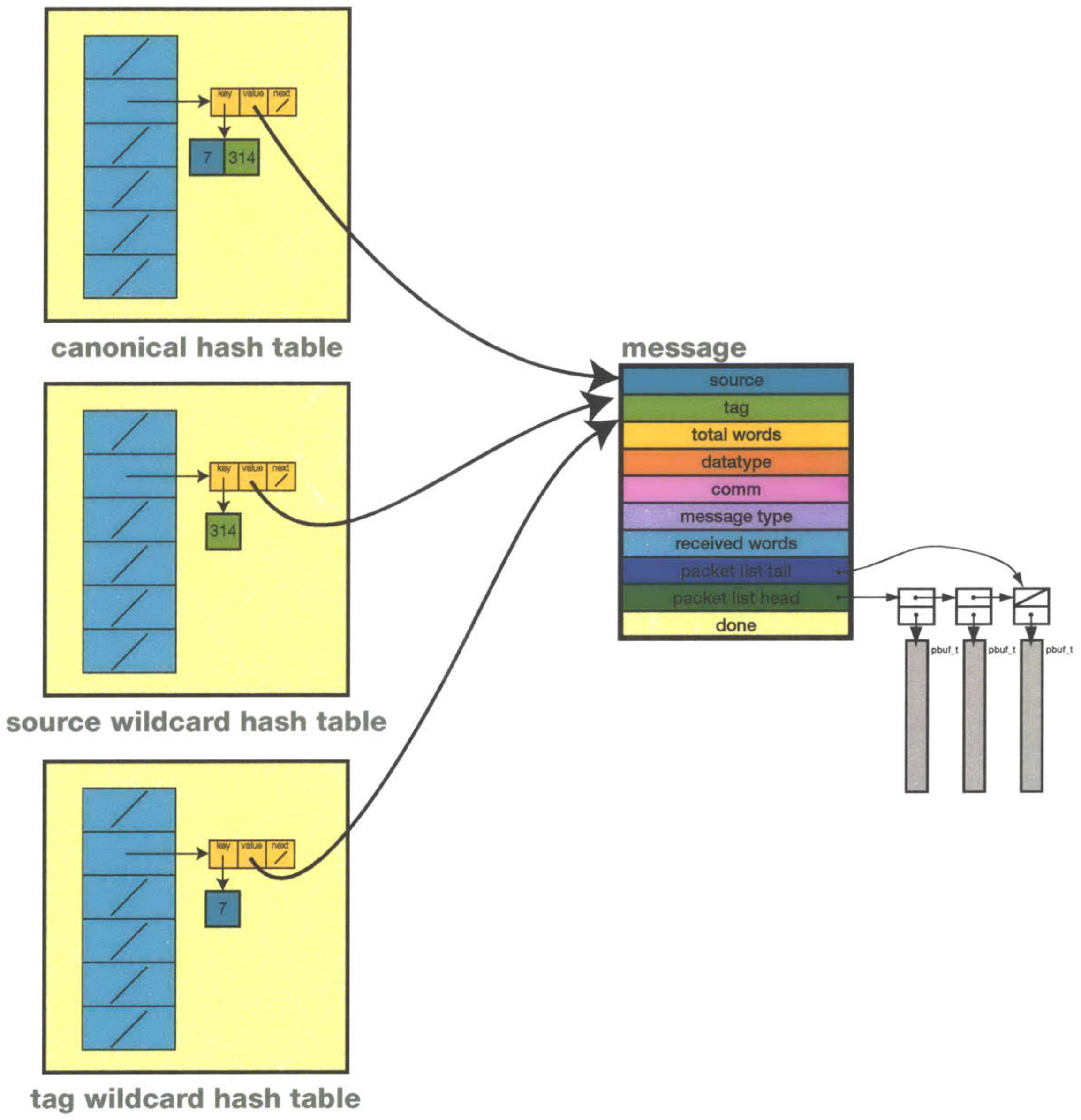


Figure 3-10: Outstanding Message Structure for Version II, showing the canonical hash table, source wildcard hash table, and tag wildcard hash table. All three hash tables point have exactly one entry, which all points to the same message. In this example, the message has a source of 7 and a tag of 314.

Since the hash tables contain pointers to message objects, and not the objects themselves, only the underlying message object must be modified as needed upon packet insertion – the hash tables do not have to be updated at all. However, when a message is consumed by the receiver, the message object pointer is removed from all three hash tables.

The three-hash-table implementation handles the no-wildcard or one-wildcard case, but does not handle the case in which both a source *and* tag wildcard are specified. To handle this case, the canonical hash table was extended to allow for removing *any* complete message. The lookup function simply traverses the list of messages until it finds one that has been completely received. Of course, when a value is removed from the canonical hash table, it must also be removed from the source wildcard and tag wildcard hash tables.

As Version II relies solely on the interrupt handler to receive data, the actions the receiver performs upon a call to `MPI_Recv` are quite different than those in Version I. When the receiver makes the `MPI_Recv` call, it looks in the OMS to see if the desired message has already arrived. If both the source and tag were specified, it looks in the canonical hash table; if the source was wildcarded and the tag was specified, it looks in the source wildcard hash table, and in the tag wildcard hash table if a tag wildcard was specified. If a matching message is returned from the lookup, the receiver checks the `done` field of the message object. If the `done` field is TRUE, the payload data is copied sequentially from the packet buffers into the user's buffer, the message is removed from all three hash tables and destroyed, and the packet buffers are reallocated to the buffer pool. If the `done` field is FALSE, or the message lookup returns NULL, the receiver simply spins in a loop, continuously checking to see if the desired conditions become satisfied. Of course, assuming a well-written user program, at some point the desired message will arrive, the interrupt will fire, the data will be read into the appropriate message object, the `done` field will be written as TRUE, the interrupt handler will return, and the desired conditions will become true. The receiver will then copy the payload and remove the message from the OMS, as described above. Thus, it is clear that in Version II, the complexity of the receiving process is in the interrupt handler, and the `MPI_Recv` simply copies and cleans up when the message has already arrived.

For reasons that will become clear in Section 3.3.3, Version II's receive implementation actually consists of two steps. The top-level `MPI_Recv` wrapper function verifies that arguments are appropriate and performs other tasks that are specific to the `MPI_Recv` function. The lower-level `_Recv` function interacts with the OMS and handles all data transportation. In fact, `_Recv` was implemented in such a way as to be a general interface for receiving messages, no matter if they were

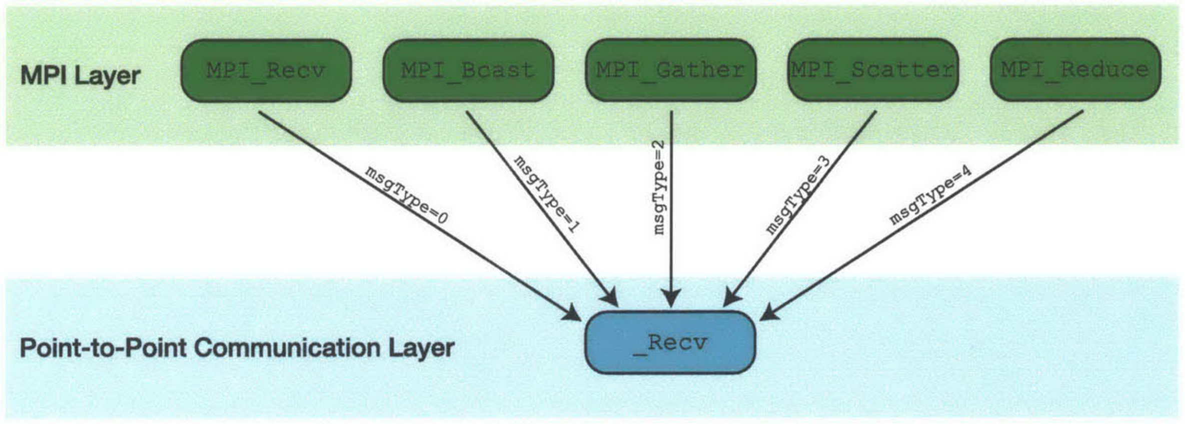


Figure 3-11: The hierarchy of various functions that receive data. High-level functions in the MPI layer parse and error-check arguments, and the `_Recv` function, part of the point-to-point communication layer, processes individual point-to-point communications, including all of the data structure management and data transportation.

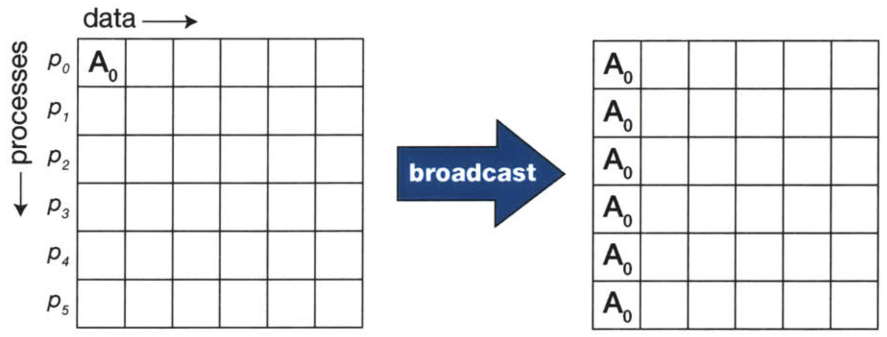
initiated by a normal `MPI_Recv`, or a collective communication function such as `MPI_Gather`. The architecture of this design can be seen in Figure 3-11, which can be generalized to an MPI layer and a point-to-point communication layer. This generalization is important when this design is used to implement collective communication functions, which are discussed in the following section.

3.3.3 Extended functionality

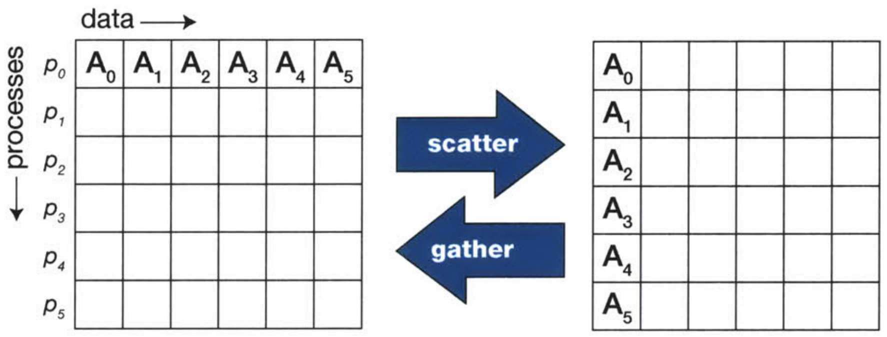
In addition to basic blocking sends and receives, Version II supports some collective communications. In particular, it supports the following operations: `MPI_Bcast`; `MPI_Gather`; `MPI_Scatter`; `MPI_Reduce`. In this Section, the design and implementation of each of these functions is briefly discussed. Note that the term ***rank*** refers to the numerical id of a process within the context of a communicator, and the term ***root*** refers to the "master" of a collective communication action (the role of the master depends on which collective communication action is being performed).

Broadcast

```
int MPI_Bcast (void *buffer,            /* starting address of buffer */
               int count,               /* number of entries in buffer */
               MPI_Datatype datatype,   /* data type of buffer */
               int root,                /* rank of broadcast root */
```



(a) `MPI_Bcast`



(b) `MPI_Gather` and `MPI_Scatter`

Figure 3-12: Collective move functions illustratred for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the item A_0, but after the broadcast all processes contain it. *Figure adapted from [13].*

```
            MPI_Comm comm)           /* communicator */
```

`MPI_Bcast` broadcasts a message from the process with the rank `root` to all processes of the group specified by `comm`, itself included. Upon return, the contents of `root`'s communication buffer has been copied to all processes. This operation is depicted in Figure 3-12(a). In Version II's implementation, the root process simply loops through a $N - 1$ send operations with a message type of `bcast`, while the receiving processes invoke a receive operation with a message type of `bcast`.

Gather

```
int MPI_Gather
```

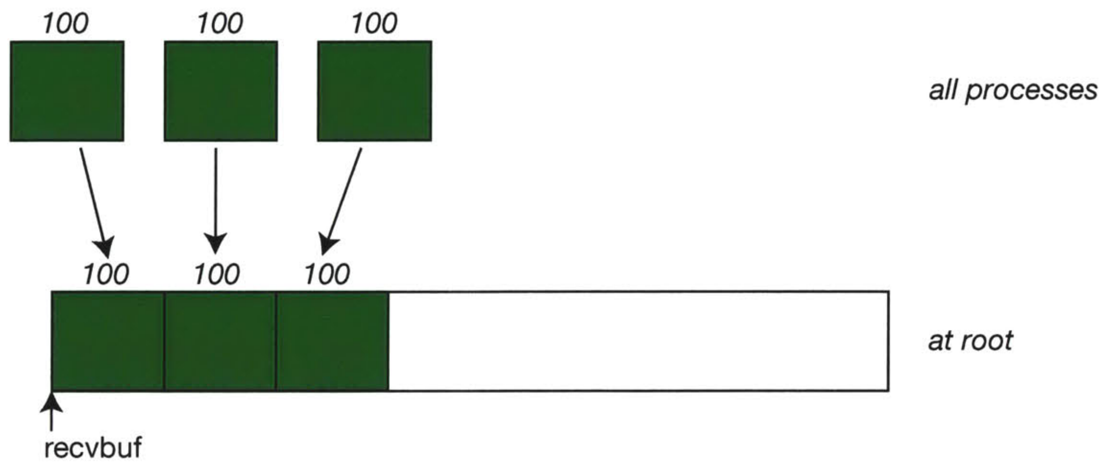


Figure 3-13: The root process gathers 100 `int`s from each process in the group. *Figure adapted from [13].*

```
(void *sendbuf,          /* starting address of send buffer */
 int sendcnt,            /* number of elements in send buffer */
 MPI_Datatype sendtype, /* data type of send buffer elements */
 void *recvbuf,          /* address of receive buffer  */
 int recvcount,          /* number of elements for any single receive */
 MPI_Datatype recvtype, /* data type of recv buffer elements */
 int root,               /* rank of receiving process */
 MPI_Comm comm)          /* communicator */
```

When `MPI_Gather` is invoked, each process in the group specified by `comm` (root process included) send the contents of its send buffer to the root process. The root process receives and assembles the messages in such a way that the resulting message received is the concatenation, in rank order, of the incoming messages. The receive buffer is ignored for all non-root processes. This operation is depicted in Figure 3-12(b). In Version II's implementation, the non-root processes simply send their message to the root, specifying a message type of `gather`. The root process loops through the $N-1$ non-root processes, concatenating them together into effectively one long message. Figure 3-13 shows an example of a root process gathering 100 `int`s from each process in the group.

Scatter

```
int MPI_Gather
   (void *sendbuf,          /* starting address of send buffer */
    int sendcnt,            /* number of elements sent to each process */
    MPI_Datatype sendtype,  /* data type of send buffer elements */
    void *recvbuf,          /* address of receive buffer */
    int recvcount,          /* number of elements in receive buffer */
    MPI_Datatype recvtype,  /* data type of recv buffer elements */
    int root,               /* rank of sending process */
    MPI_Comm comm)          /* communicator */
```

`MPI_Scatter` is the inverse operation of `MPI_Gather`. The result of this operation is *as if* the root executed N send operations, sending a 1/N portion of the send buffer to each of the different non-root processes, with the ith segment going to the ith process in the group. Each process receives the message that was sent into their `recvbuf`. This operation is depicted in Figure 3-12(b). In Version II's implementation, the root process sends messages segment-by-segment to each of the processes in the group with a message type of `scatter`. The non-root processes enact a receive with a message type of `scatter`. Figure 3-14 shows an example of a root process scattering 100 `ints` to each process in the group.

Reduce

```
int MPI_Reduce (
  void *sendbuf,          /* address of send buffer */
  void *recvbuf,          /* address of receive buffer */
  int count,              /* number of elements in send buffer */
  MPI_Datatype datatype,  /* data type of elements of send buffer */
  MPI_Op op,              /* reduce operation */
  int root,               /* rank of root process */
  MPI_Comm comm )         /* communicator */
```

`MPI_Reduce` performs a global reduction operation using a function such as sum, max, logical AND, etc. across all members of a group. More specifically, it combines the elements provided in the

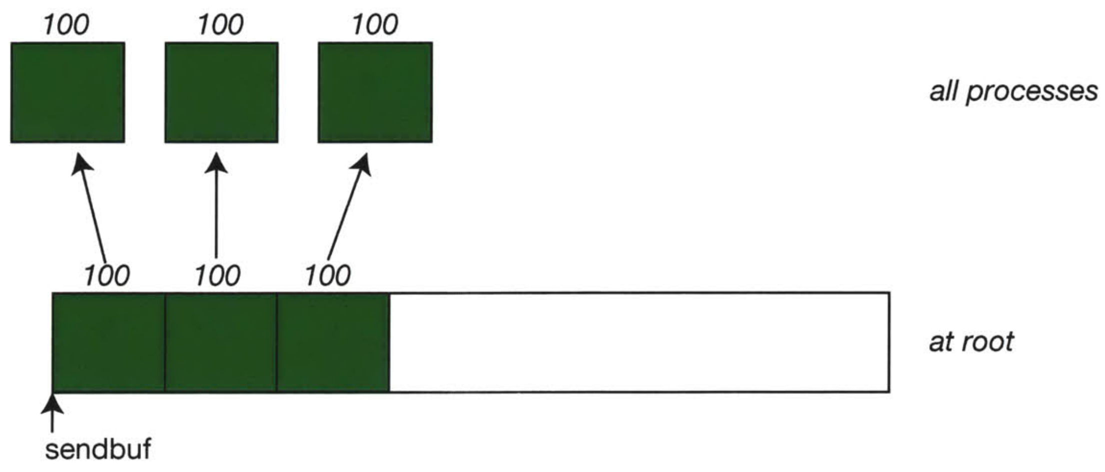


Figure 3-14: The root process scatters 100 `int`s to each process in the group. *Figure adapted from [13].*

input buffer of each process in the group, using the operation `op`, and returns the combined value in the output buffer of the process with rank `root`. Figure 3-15 depicts this operation.

Version II implements the reduction operation efficiently, using a binary tree reduction approach, as opposed to the straightforward linear approach. Figure 3-16 contrasts the two approaches, showing 8 processes performing a sum reduction operation. In most cases, the binary approach is superior to the linear approach because it parallelizes computation and also decreases the message congestion at the root node. In the linear approach, seen in Figure 3-16(a), the sum is computed after 7 messages are sent to the root, where *all* computation is performed. Contrastingly, in the binary approach, seen in Figure 3-16(b), the sum operation is performed in $\log 8 = 3$ stages, spread out over numerous processes. Finally, at time 3, the root receives the final message and does one final addition. Again, as the computation is spread out over multiple processing elements, the binary approach achieves higher performance in most cases, relative to the linear approach.

Version II supports the following reduction operations: max; min; sum; product; logical AND; binary AND; logical OR; binary OR; binary XOR. The MPI Standard defines a larger set of predefined operations, and also allows users to create their own custom reduction operations, but Version II does not support functions other than those just listed.

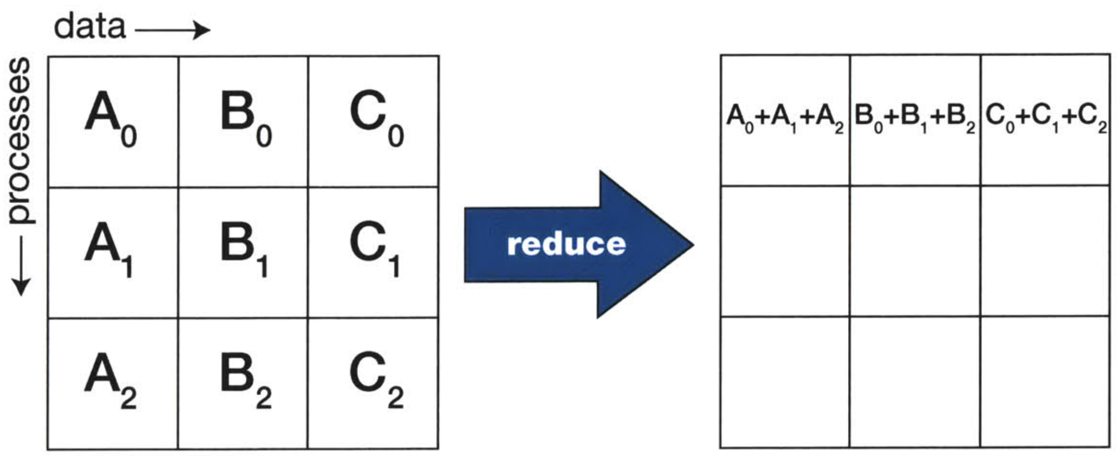


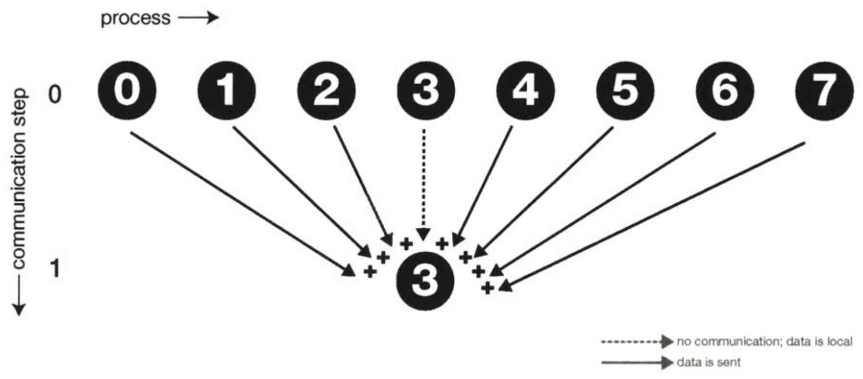
Figure 3-15: Reduce function illustrated for a group of three processes. Each row of boxes represents data items in one process. Thus, initially each process has three items; after the reduce the root process has three sums. *Figure adapted from [13].*

3.3.4 Analysis of Version II

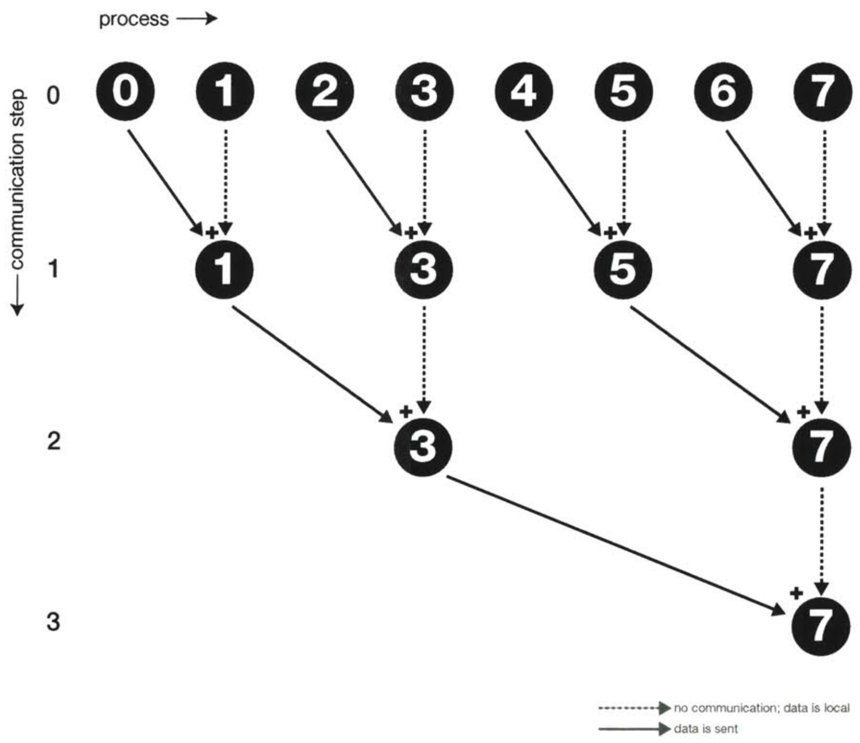
Version II provided a significantly improved rMPI implementation relative to Version I. Overall, it met the design goals outlined in the beginning of this section. Firstly, the interrupt-driven design proved to be a prudent way to architect the system, allowing for a robust design that is immune to the deadlock situation described in Section 3.2.3, and overall made for a cleaner design. The completely overhauled Outstanding Message Structure transformed message insertions, lookups, and removals from linear to constant time. Furthermore, this new design proved to be cleaner and more extensible. For instance, this extensibility allowed easy additions of new features such as the `msgType` field, which in turn allowed higher level MPI functions, such as `MPI_Bcast`.

To verify correct operation, a number of tests were run. These tests included all of those from Version I, as well as new tests that stressed the system's new features, especially the new higher level functions and tests that made Version I deadlock. All of the tests proved successful.

It is clear that Version II's design and implementation was a significant improvement over that of Version I. However, relative to a full MPI implementation, Version II still only implements a small subset of the MPI standard. During the implementation of Version II, it became clear that the architecture of rMPI was very similar to the architecture of full MPI implementations. Implementations such as MPICH [33] and the LAM/MPI [10, 28] were referenced while the higher-level functions



(a) Linear Reduction (Process 3 shown as root for clarity)



(b) Binary Reduction (Process 7 shown as root)

Figure 3-16: This figure compares linear and binary reduction algorithms. Solid arrows represent data being sent by means of a message, while a dashed arrow represents progress in the computation, but no data is sent since it is already local to that process. In the linear case, the sum is computed after 7 messages are sent to the root, where *all* computation is performed serially. Contrastingly, in the binary approach, the sum operation is performed in $\log 8 = 3$ stages, spread out over numerous processes. Finally, at time 3, the root receives the final message and does one final addition. As the computation is spread out over multiple processing elements, when the computation is expensive the binary approach usually achieves higher performance relative to the linear approach.

of Version II were implemented. These full implementations certainly provided helpful guidance, as well as confirmation that the architecture of Version II was well-designed, as it turned out to mimic the full implementations in some key ways. For instance, the receive hierarchy, as shown in Figure 3-11, was very similar to that of both the MPICH and LAM/MPI architecture. Version II was intended to be the final production implementation of rMPI, with enhancements and additional functionality added incrementally as needed. However, in light of the similarities in architecture to MPICH and LAM/MPI, and given the small subset of MPI that Version II provided, it became apparent that it made sense to build yet another version of rMPI that leveraged already-existing implementations more fully. Version III, the third incarnation of rMPI, strives toward this goal, and is discussed next, in Chapter 4.

Chapter 4

Design and Implementation of Production System

This chapter builds on what was learned from the prototype systems discussed in Chapter 3, and describes the third and final implementation of rMPI. When compared to the first two versions, the third and ultimate version of rMPI is superior in many ways. Largely the result of lessons learned both from Version I and Version II, as well as from a complete high-performance MPI implementation, Version III is, by far, the best performing and most complete MPI implementation of the three. As mentioned in Section 3.3.4, Version II's design was solid in many ways, and even mimicked that of complete production-quality MPI implementations. As development on Version II continued, it became clear that the decision to continue to add new functionality incrementally was somewhat inefficient, as much of the implementation of the new functionality was derived from already-existing MPI implementations. Thus, the decision was made to integrate Version II with a high-quality open source MPI implementation, the result of which became known as Version III.

In reality, Version III was almost essentially all new code; it did not truly utilize the contents of Version II directly. However, the high-level architecture, especially the notions of interrupt-driven design and the outstanding message structure, were definitely integrated into the new design. Thus, Version III was an all-new implementation, but was certainly related to Version II (and Version I) in spirit. Ultimately, Version III strove to implement a large subset of the MPI standard to allow users to run any MPI program they desire. Furthermore, it strove to be efficient, robust, and extensible. As is described in this section, Version III respectably achieved these goals.

As was mentioned above, Version II's design was similar to MPICH and LAM/MPI, especially in the way that receives were separated into multiple abstract levels, as depicted in Figure 3-11. Thus, the goal became to conflate Version II with a production-quality MPI implementation. The first step toward reaching this goal was to decide which MPI implementation was best suited for the task. Multiple implementations were examined, but it quickly became clear that either MPICH [33], or LAM/MPI [10, 28] were most likely to meet the requirements outlined at the start of this chapter because their designs are somewhat reminiscent of Version II, and, perhaps more importantly, their implementations are high performance and complete.

MPICH, one of the most popular MPI implementations, is intended to be a portable MPI implementation whose development began at the same time as the MPI definition process itself in the early 1990s. MPICH's goal is to give users a free, high-performance MPI implementation that is usable on a variety of platforms in order to support the adoption of the MPI Standard. It also strives to aid vendors and researchers in creating their own implementations that are customized and optimized for their platforms. To achieve these goals of portability and performance, MPICH relies on a central mechanism called the abstract device interface (ADI). The ADI is a set of function definitions (usually either C functions or macro definitions) in terms of which the user-callable standard MPI functions may be expressed. The ADI is in charge of packetizing messages and assembling appropriate headers, matching posted receives with incoming messages, managing multiple buffering policies, and handling heterogeneous communications [33]. While this abstraction could, in theory, have lent itself well to melding with Version II, a closer look at the underlying ADI proved to be quite cumbersome and inefficient. Thus, MPICH was bypassed in search of something better.

LAM/MPI is another production quality MPI implementation, which is generally known to be one of quite high-quality in the supercomputing community. LAM/MPI is an open-source implementation, including all of the MPI-1.2 and much of the MPI-2 standards, and is intended for production as well as research use. Like MPICH, LAM/MPI is also intended to perform well on a variety of platforms, from small off-the-shelf single CPU clusters to large SMP machines with high speed networks, and designed in such a way as to be extensible to such a variety of platforms. LAM/MPI's software architecture is structured according to the so-called LAM System Services Interface (SSI), a two-tiered modular framework for collections of system interfaces. The overall intent for SSI is to allow a modular "plug-n-play" approach to the system services that LAM and MPI must use [17]. Such an architecture facilitates development of modules that allow LAM/MPI to run on many parallel computer systems. Upon close inspection of the SSI and other components of the LAM/MPI architecture, through both its documentation and source code, it became clear that it was the most appropriate choice of full MPI implementations to use to assist in building Version III. This section describes how portions of LAM/MPI were incorporated to achieve the resulting Version III implementation of rMPI.

Due to reasons stemming from Raw's virtually non-existent operating system, LAM/MPI was used in a somewhat untraditional way. In order to understand the changes that needed to be made to LAM, and how it was integrated, it must first be thoroughly understood in its native form. Section 4.1 expounds the LAM/MPI architecture.

4.1 LAM/MPI

LAM, or Local Area Multicomputer, is an open source implementation of the Message Passing Interface developed and maintained at Indiana University. LAM/MPI includes a full implementation of the MPI-1 standard and much of the MPI-2 standard, consisting of over 150 directories and nearly 1,400 files of source code. Because of its large and complex nature, extending or even understanding the LAM implementation is typically quite challenging. To this end, LAM developers created a lightweight component architecture that is specifically designed for high-performance message passing and makes it possible to write and maintain small, independent component modules that integrate into the LAM framework. LAM is generally intended to run on systems with operating systems, such as a cluster of workstations running Linux.

As shown in Figure 4-1, LAM/MPI has two primary components, the LAM runtime environment (RTE) and the MPI communications layer, which both interact with the operating system. The following sections discuss these layers in detail.

4.1.1 LAM Layer

The LAM layer presides over the LAM RTE, and allows other layers to interact with the RTE through C API functions. The RTE contains user-level daemons that are used for interprocess communication, process control, remote file access, and I/O forwarding. The RTE is an integral part of the LAM system, and often acts as the liaison between the MPI layer and the operating system through the use of user-level daemons. The RTE's daemons must be running before a user starts an MPI program [28].

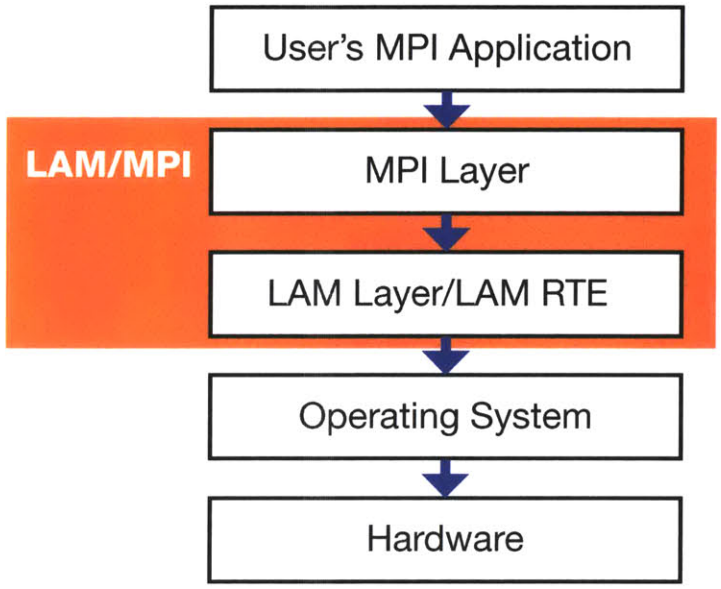


Figure 4-1: LAM/MPI's high-level architecture. The layers include the user's MPI application, the MPI Layer, the LAM layer, and the underlying operating system.

4.2 MPI Communications Layer

The MPI layer implements the actual MPI functionality, from the user-visible MPI API, down to the low-level routines that actually move data and manage internal data structures. This layer is actually sub-divided into multiple components and layers, as seen in Figure 4-2. The upper layer constitutes the standardized MPI API function calls and associated bookkeeping logic, and acts as the interface to the lower-level components. The lower layer contains a number of different interchangeable components that implement various lower-level routines, including the point-to-point communication components, the collective communication components, the checkpoint/restart components, and the boot components.

4.2.1 MPI Point-to-Point Communication Components

The Request Progression Interface (RPI) provides the point-to-point communication routines that the upper layer's MPI routines use to actually move messages from one process to another. Components that plug into this interface accept MPI messages from the MPI layer and push them down to the destination MPI process in the case of a send. Likewise, in the case of a receive, such components also accept messages from peer MPI processes and forward them up to the MPI layer in response

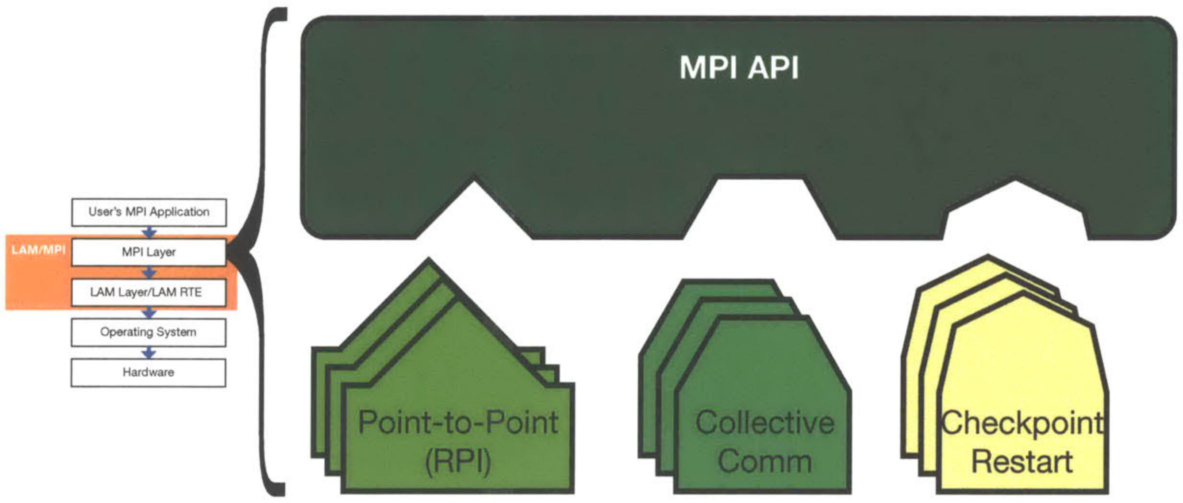


Figure 4-2: The MPI Communications Layer of LAM/MPI. The high-level MPI layer implements the MPI API's public interfaces, which are implemented with back-end components, including the point-to-point components, the collective communication components, and the checkpoint/restart components.

to a matching receive request. The MPI layer is oblivious to how messages propagate from process P_i to process P_j; the abstraction barrier between the MPI layer and the RPI hides these details. The MPI layer is only aware of the current state of its requests for sends and receives (created, started, completed). Thus, the RPI module has complete freedom in its implementation, and can choose to transport messages from source to destination in any means it deems appropriate, most likely concentrating on robustness and high-performance for a given platform. The MPI layer handles higher level operations such as buffer packing/unpacking and message data conversion, while the RPI is in charge of initializing data transport connections between processes, transporting messages across the connections, message synchronization, cleanup, and maintenance of proper state [26, 27].

Different RPI implementations can be interchanged depending on how data is to be transported. LAM ships with a number of different already-implemented RPIs, including those that support data migration via shared memory, TCP, InfiniBand [2], and Myrinet [5]. The variety of platforms for which new RPIs can be written is quite expansive, as the RPI API was written generally enough to support future computer systems. Version III employs a new RPI developed specifically for the Raw General Dynamic Network. The design and implementation of the Raw GDN RPI is discussed below.

4.2.2 MPI Collective Communication Components

The MPI Collective Communication Components provide implementations for the MPI collective algorithms that perform operations such as broadcasting, spreading, collecting, and reducing. These components are typically implemented out of underlying point-to-point functions for message passing, but can be written using a variety of algorithms. For instance, one collective communication component may implement `MPI_Bcast` as a `for` loop of `MPI_Sends`, while another may use a binary tree distribution algorithm. Similarly, the algorithm that is employed may be dependent on the number of processes that are involved in the communication. For instance, Figure 3-16 shows two different strategies for reduction, linear and binary tree. A well-constructed collective communication component which takes the underlying platform and communication technology and topology into account, may choose, say, the linear algorithm for $N \leqslant 4$ processes, and the binary tree algorithm for $N > 4$ processes. In general, the LAM designers made the component interface sufficiently generic to allow this component's design to be customized for a particular platform. rMPI Version III used a modified version of the `lam_basic` collective communication module, which is included in the LAM/MPI distribution. Its implementation is outlined below.

4.2.3 Checkpoint/Restart Components

LAM/MPI supports checkpointing and restarting through the Checkpoint/Restart component. The primary abstract actions that are supported by this module are **checkpoint**, **continue**, and **restart**. This component requires that all MPI SSI modules, including the RPI, support the ability to checkpoint and restart themselves [24, 28]. The current implementation of rMPI Version III does not support checkpointing, as the Raw GDN RPI does not support it.

4.2.4 Boot Components

The boot components are used to start and expand the LAM RTE. They are generally used for such actions as identifying remote hosts, starting processes on remote nodes, and bootstraping newly started processes by sending them initialization protocol information. As the LAM RTE was removed in rMPI Version III, no boot components exist in Version III.

4.3 Version III Architecture

In light of the above overview of LAM/MPI, the design and implementation of Version III is now presented. The end goal of the Version III implementation of rMPI was to leverage as much of the LAM/MPI infrastructure as possible to build a robust MPI implementation for the Raw architecture. As discussed earlier, LAM/MPI's architecture relies on the existence of an operating system that supports concurrent processes and interprocess communication to support its integral runtime environment. In addition to implementing Raw-specific components, such as the RPI, which deals with the point-to-point message transportation processes, since Raw lacks operating system support, much of the non-component core LAM implementation had to be modified. Figure 4-3 shows both the canonical LAM/MPI architecture, and the resulting rMPI Version III architecture. While LAM typically requires many processing threads to run, the Version III implementation could only support one thread. Therefore, much of the functionality that the LAM RTE system provided was removed in Version III, and some of it was injected into the single thread. For instance, the following LAM RTE facilities were removed when transitioning to a single-threaded system:

- Boot components
- Tracing functionality
- Checkpointing/restarting functionality
- kernel messages

The details of the individual components of Version III are discussed next.

4.3.1 High-Level MPI Communications Layer

As mentioned above, the MPI Communications Layer implements the actual MPI functionality, from the user-visible MPI API, down to the low-level routines that actually implement and manage internal data structures. The high-level MPI communications layer refers to the components that implement the actual user-visible MPI API, such as the top-level call to `MPI_Send` or `MPI_Comm_size`. In Version III, most of the high-level MPI layer consists of over 250 source code files directly adapted from the LAM/MPI source tree. In most cases, these contain wrapper functions that accept MPI API calls from the user-level MPI program, check arguments for conformance to the Standard, handle

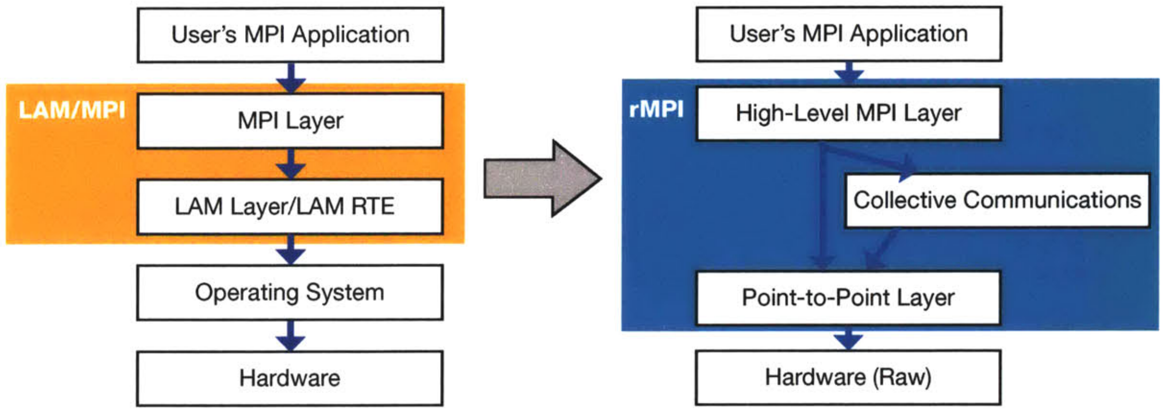


Figure 4-3: Comparison of LAM/MPI Architecture to rMPI Version III Architecture. Both architectures have the user's MPI application as the top-level and the hardware platform as the bottom-most level. Version III removes the LAM Layer/LAM RTE and does not rely on any operating system support. For clarity, the three main components of Version III are shown; they are the high-level MPI layer, the collective communications layer, and the point-to-point layer.

errors appropriately if necessary, process trivial cases, and finally call the next-lower-level routine that actually implements, potentially, the movement of data.

To illustrate the high-level MPI communications layer, consider an example. Figure 4-4 shows the almost true-to-life[1] source code taken from the Version III implementation. The initial two calls to `lam_initerr_m` and `lam_setfunc_m` (lines 6-7) are part of the LAM/MPI updown interface, used for runtime debugging. The updown interface keeps track of the current call stack, and returns detailed error reports upon a runtime error. Next, on lines 9-12, conditions for the `tag` parameter are checked, and an error is returned if appropriate. Next, some trivial send cases are checked. If the specified destination process is `MPI_ANY_SOURCE`, an MPI constant that is only appropriate when receiving, an error is thrown. Next, on lines 20-21, if the destination is `MPI_PROC_NULL`, another MPI constant specifying, as the name implies, a NULL process handle, `MPI_Send` simply returns successfully, having sent nothing. Next, the function checks to see if the RPI, the underlying point-to-point module, contains a "fastsend" implementation (a concept that is discussed below), which the Raw GDN RPI does have. Another trivial error check is performed on line 25; if the communication context is `MPI_COMM_NULL`, an error is returned. Finally, on line 30, the underlying RPI fastsend function is called. This function is implemented in the RPI, and handles all of the details of sending the message. Note that the parameters that were passed to `MPI_Send` are simply forwarded onto the

[1]The actual `MPI_Send` code is broken down into two files; for clarity, the two files were coalesced into one.

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
                        int dest, int tag, MPI_Comm comm)
{
        int             err;

        lam_initerr_m();
        lam_setfunc_m(BLKMPISEND);

        if (tag < 0 || tag > lam_mpi_max_tag) {
                return(lam_err_comm(comm, MPI_ERR_TAG, EINVAL,
                                                "out of range"));
        }

        /*
         * Handle the trivial cases.
         */
        if (dest == MPI_ANY_SOURCE)
                return(lam_mkerr(MPI_ERR_RANK, EINVAL));

        if (dest == MPI_PROC_NULL)
                return(MPI_SUCCESS);

        if (RPI_HAS_FASTSEND(comm, dest) && reqtype == LAM_RQISEND &&
            lam_rq_nactv == 0) {
            if (comm == MPI_COMM_NULL)
                        return(lam_mkerr(MPI_ERR_COMM, EINVAL));

                errno = 0;

                err = RPI_FASTSEND(buf, count, dtype, dest, tag, comm);

                if (err == MPI_SUCCESS ||
                        (err != MPI_SUCCESS && errno != EBUSY)) {

                        return(err);
                }
        }

        if (err != MPI_SUCCESS) return(lam_errfunc(comm, BLKMPISEND, err));

        lam_resetfunc_m(BLKMPISEND);
        return(MPI_SUCCESS);
}
```

Figure 4-4: Top-level MPI_Send code. Performs error checking, handles trivial cases if applicable, and calls lower-level point-to-point send routine.

RPI_FASTSEND function, the details of which are discussed below. Finally, after returning from RPI_FASTSEND, MPI_Send checks the returned error code, removes the MPI_Send function from the updown interface (as it is about to leave this function), and returns. The returned error code complies with that specified in the MPI Standard.

While the above example detailed MPI_Send, most of the functions in the high-level MPI layer have this form. This layered design allows underlying RPIs and collective operations components to be interchanged without rewriting the common higher layer. While, as mentioned above, a large portion of this layer was retained from the LAM/MPI implementation, sections of the implementation were removed or modified. For instance, all interactions with the LAM RTE, including those dealing with checkpointing and tracing, were removed for the Version III implementation.

User-Defined Datatypes and Packing

In addition to allowing the user to send and receive a sequence of elements that are contiguous in memory, the MPI communication mechanisms support sending of non-homogeneous data, or that which is not contiguous in memory. Users can define derived datatypes, and use them in place of the predefined datatypes. The high-level MPI layer implements functions dealing with creation and use of derived datatypes. These functions include, but are not limited to, the following:

- Datatype accessors (*e.g.*, MPI_Type_get_extent)
- Datatype constructors (*e.g.*, MPI_Type_contiguous)
- Vector datatype constructors (*e.g.*, MPI_Type_vector)
- Indexed datatype constructors (*e.g.*, MPI_Type_indexed)
- Block indexed datatype constructors (*e.g.*, MPI_Type_create_indexed_block)
- Structure datatype constructors (*e.g.*, MPI_Type_create_struct)
- Datatype management (*e.g.*, MPI_Type_commit)
- Datatype resizing (*e.g.*, MPI_Type_create_resized)
- Datatype packing and unpacking (*e.g.*, MPI_Pack, MPI_Unpack)

MPI_All_gather	MPI_All_gatherv	MPI_All_reduce
MPI_Alltoall	MPI_Alltoallv	MPI_Alltoallw
MPI_Barrier	MPI_Bcast	MPI_Gather
MPI_Gatherv	MPI_Reduce	MPI_Reduce_scatter
MPI_Scan	MPI_Scatter	MPI_Scatterv

Table 4.1: MPI Collective Operations, all implemented in Version III.

Version III implements the user-defined datatypes and packing functions, ported directly from the LAM/MPI implementation. See [13] for more information on user-defined datatypes and packing.

Communicators

Communication domains, or communicators, are an important MPI abstraction that allow one to subdivide the processes into groups, each of which can perform independent work. The collective operations are defined for communicators, so one could call `MPI_COMM_BCAST` on a communicator `c` that is a subset of `MPI_COMM_WORLD`, and the broadcast would occur only to members of the communicator `c`. The high-level MPI layer implements functions dealing with the management of communicators, which include, but are not limited to, the following:

- Communicator accessors (*e.g.,* `MPI_Comm_size`, `MPI_Comm_rank`)
- Communicator constructors (*e.g.,* `MPI_Comm_split`)
- Communicator destructors (*e.g.,* `MPI_Comm_free`)

Version III implements the communicator management functions, ported from the LAM/MPI implementation. See [13] for more information on user-defined datatypes and packing.

4.3.2 MPI Collective Communications Components

As described above, the MPI Collective Communications Components are interchangeable modules that provide implementations for the MPI collective algorithms that provide operations such as broadcasting, scattering, gathering, and reducing. As mentioned above, these components can be optimized for a particular platform or network topology to take certain system-specific characteristics into account.

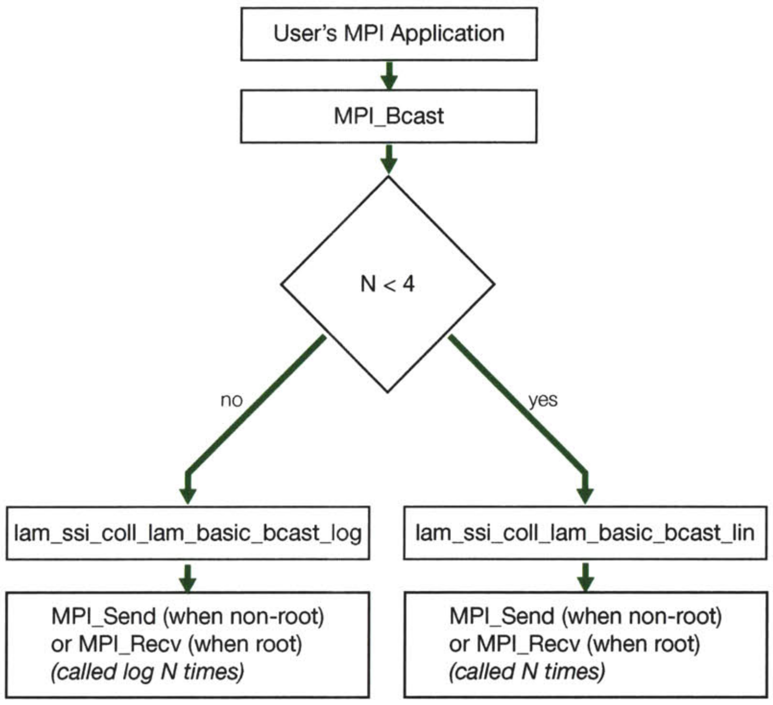


Figure 4-5: Function control flow upon call from user's MPI application to `MPI_Bcast`. `MPI_Bcast` checks for argument errors and handles trivial cases, and then calls a small function which determines which function in the collective communications module to call depending on the size of the caller's communicator. The actual broadcast implementations execute either a linear or logarithmic broadcast, using regular `MPI_Send`s and `MPI_Recv`s to send individual point-to-point messages.

Version III used a modified version of the `lam_basic` module, which is included in the LAM/MPI distribution. The `lam_basic` module is a full implementation of the MPI collective functions (see Table 4.1 for a list of the MPI collective functions), built out of the underlying point-to-point communications layer. As with the high-level MPI communications layer, the checkpointing and tracking functions were removed from the implementation on account of the lack of LAM RTE.

To illustrate how the `lam_basic` module works, consider the example of broadcasting. Figure 4-5 shows the control flow upon a user's call to `MPI_Bcast`. `MPI_Bcast`, shown in Figure 4-6, is implemented in the high-level MPI communications layer, and looks quite similar to `MPI_Send`, seen in Figure 4-4. `MPI_Bcast` performs error checking on arguments, handles trivial cases, and then calls the broadcast function of the collective communications module. The broadcast function pointer is extracted from the collective communications module on line 12, `func = comm->c_-ssi_coll.lsca_bcast`, and is later invoked in line 55. `func` analyzes the number of processes in the calling communicator comm and calls one of two broadcast functions, lam_ssi_coll_lam_basic_bcast_l the linear broadcast algorithm, or lam_ssi_coll_lam_basic_bcast_log, the logarithmic broadcast algorithm. As seen in Figure 4-5, lam_ssi_coll_lam_basic_bcast_lin is called when $N \leqslant 4$ processes are in comm, and lam_ssi_coll_lam_basic_bcast_log is called when $N > 4$ processes are in comm. The linear and logarithmic implementations of the two broadcast algorithms can be seen in Figure 4-7 and Figure 4-8, respectively. Note how both implementations are built out of MPI_Send and MPI_Recv primitives; this is typically the case with most of the collective communications algorithms. In fact, sometimes they are built out of each other: MPI_Allreduce is simply an MPI_Reduce followed by an MPI_Reduce. Overall, the lam_basic module allowed straightforward integration of the MPI collective algorithms through a design that mimicked that of Version II's collective algorithms.

```
int MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm) {
    int size, err;
    struct _gps *p;
    lam_ssi_coll_bcast_fn_t func;

    lam_initerr(); lam_setfunc(BLKMPIBCAST);

        if (comm == MPI_COMM_NULL) {
                return(lam_errfunc(comm, BLKMPIBCAST,
                                        lam_mkerr(MPI_ERR_COMM, EINVAL)));
        }
    func = comm->c_ssi_coll.lsca_bcast;
    if (datatype == MPI_DATATYPE_NULL) {
                return(lam_errfunc(comm, BLKMPIBCAST,
                                        lam_mkerr(MPI_ERR_TYPE, EINVAL)));
    }
    if (count < 0) {
                return(lam_errfunc(comm, BLKMPIBCAST,
                                        lam_mkerr(MPI_ERR_COUNT, EINVAL)));
    }
    if (LAM_IS_INTRA(comm)) {
                MPI_Comm_size(comm, &size);
                if ((root >= size) || (root < 0)) {
                        return(lam_errfunc(comm, BLKMPIBCAST,
                                                lam_mkerr(MPI_ERR_ROOT, EINVAL)));
                }
                p = &(comm->c_group->g_procs[root]->p_gps);
                lam_setparam(BLKMPIBCAST,
                                root | (p->gps_grank << 16),
                                (p->gps_node << 16) | p->gps_idx);

                if (count == 0 && comm->c_ssi_coll.lsca_bcast_optimization) {
                        lam_resetfunc(BLKMPIBCAST);
                        return(MPI_SUCCESS);
                }
                /* If there's only one node, we're done */
                else if (size <= 1) {
                        lam_resetfunc(BLKMPIBCAST);
                        return(MPI_SUCCESS);
                }
        } else {
                MPI_Comm_remote_size(comm, &size);
                if (!(((root < size) && (root >= 0))
                                || (root == MPI_ROOT) || (root == MPI_PROC_NULL))) {
                        return(lam_errfunc(comm, BLKMPIBCAST,
                                                                lam_mkerr(MPI_ERR_ROOT, EINVAL)));
                }
        }
        if (func == NULL) {
                return(lam_errfunc(comm, BLKMPIBCAST,
                                                        lam_mkerr(MPI_ERR_OTHER, ENOT_IMPLEMENTED)));
        }

        /* Call the coll SSI to actually perform the broadcast */
          if ((err = func(buff, count, datatype, root, comm)) != MPI_SUCCESS)
                return(lam_errfunc(comm, BLKMPIBCAST,
                                                        lam_mkerr(MPI_ERR_COMM, err)));

        lam_resetfunc(BLKMPIBCAST);
        return MPI_SUCCESS;
}
```

Figure 4-6: High-level MPI Layer `MPI_Bcast` code. Performs error checking, handles trivial cases if applicable, and calls lower-level routine implemented in the `lam_basic` module.

4.3.3 Point-to-Point Communication Layer

As mentioned above, the Request Progression Interface (RPI) provides the point-to-point communication routines that the upper layer MPI routines use to actually move messages from one process to another. Figure 4-4 showed how `MPI_Send` called down to `RPI_FASTSEND`, which is implemented in the point-to-point communication layer. This section describes the Raw GDN RPI, the module that implements all of the low-level details that are required for sending and receiving messages on the Raw general dynamic network.

Sending

Version III's sending protocol is quite similar to that of Versions I and II. If they are long, messages are broken up into packets and sent out on the GDN with appropriate header information. Once again, there exist two packet types in Version III, the *first packet type* and the *non-first packet type*. Figure 4-9(a) shows the format of the first packet sent in any MPI message in Version III's implementation. It contains a header, or envelope, consisting of 4 words, as described here:

grank The global rank of the sending process.

size The size of the entire MPI message, in bytes.

tag The tag of the MPI message.

cid The communication domain identifier, specifying the context in which the `rank` specified by the receiving process should be interpreted.

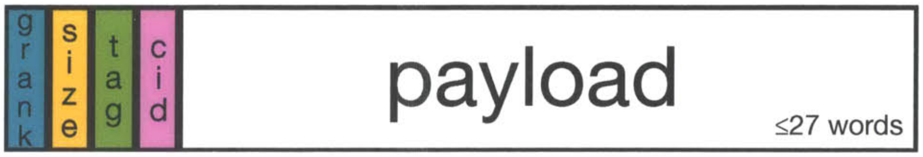
The only fields that need any clarification beyond what was mentioned in the Version I and Version II design overviews above are the `grank` and `cid` field. The `grank` field is simply the global rank, or the rank within `MPI_COMM_WORLD`. The `cid` is an integer identifying the communication domain in which `rank` should be interpreted by the receiving process. Among other things, a communicator serves to define a set of processes that can be contacted, and processes can belong to one or many communicators (by default, all processes belong to the `MPI_COMM_WORLD` communicator). For instance, process P_i may invoke a send to a process with rank 5 in communication context `MPI_COMM_WORLD` (which, in Version III's implementation, has a `cid` of 0) or to a process with a rank of 1 in communication context 1, perhaps created from a call to `MPI_Comm_split`. In some situations, these two sends may go to the same physical process, but will be treated as two distinct

```
/*
 *      bcast_lin
 *
 *      Function:       - broadcast using O(N) algorithm
 *      Accepts:        - same arguments as MPI_Bcast()
 *      Returns:        - MPI_SUCCESS or error code
 */
int lam_ssi_coll_lam_basic_bcast_lin(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
{
        int i;
        int size;
        int rank;
        int err;

        MPI_Comm_size(comm, &size);
        MPI_Comm_rank(comm, &rank);
        lam_mkcoll(comm);

        /* Non-root receive the data. */

        if (rank != root) {
                err = MPI_Recv(buff, count, datatype, root,
                                        BLKMPIBCAST, comm, MPI_STATUS_IGNORE);
                lam_mkpt(comm);
                if (err != MPI_SUCCESS)
                        return err;

                return MPI_SUCCESS;
        }

        /* Root sends data to all others. */
        for (i = 0, preq = reqs; i < size; ++i) {
                if (i == rank)
                        continue;

                err = MPI_Send(buff, count, datatype, i, BLKMPIBCAST,
                                        comm);
                if (err != MPI_SUCCESS) {
                        lam_mkpt(comm);
                        return err;
                }
        }

        /* All done */
        return (MPI_SUCCESS);
}
```

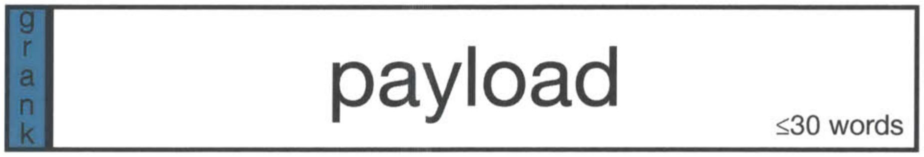
Figure 4-7: Linear implementation of the collective communications component `MPI_Bcast` implementation from the `lam_basic` module.

```
/*
 *      bcast_log
 *
 *      Function:       - broadcast using O(log(N)) algorithm
 *      Accepts:        - same arguments as MPI_Bcast()
 *      Returns:        - MPI_SUCCESS or error code
 */
int lam_ssi_coll_lam_basic_bcast_log(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
{
        int i, size, rank, vrank, peer, dim, hibit, mask, err, nreqs;

        MPI_Comm_rank(comm, &rank);
        MPI_Comm_size(comm, &size);
        lam_mkcoll(comm);
        vrank = (rank + size − root) % size;
        dim = comm−>c_cube_dim;
        hibit = lam_hibit(vrank, dim);
        −−dim;

        /* Receive data from parent in the tree. */
        if (vrank > 0) {
                peer = ((vrank & ~(1 << hibit)) + root) % size;
                err = MPI_Recv(buff, count, datatype, peer,
                                        BLKMPIBCAST, comm, MPI_STATUS_IGNORE);
                if (err != MPI_SUCCESS) {
                        lam_mkpt(comm);
                        return err;
                }
        }

        /* Send data to the children. */
        preq = reqs;
        nreqs = 0;
        for (i = hibit + 1, mask = 1 << i; i <= dim; ++i, mask <<= 1) {
                peer = vrank | mask;
                if (peer < size) {
                        peer = (peer + root) % size;
                        ++nreqs;
                        err = MPI_Send(buff, count, datatype, peer, BLKMPIBCAST,
                                                comm, preq++);
                        if (err != MPI_SUCCESS) {
                                lam_mkpt(comm);
                                return err;
                        }
                }
        }

        /* All done */
        lam_mkpt(comm);
        return (MPI_SUCCESS);
}
```

Figure 4-8: Logarithmic implementation of the collective communications component MPI_Bcast implementation from the lam_basic module.



(a) First packet



(b) Non-first packet

Figure 4-9: Packet formats in Version III implementation.

version	**first packet header (payload) length**	**non-first packet header (payload) length**
Version I	5 (26)	2 (29)
Version II	7 (24)	3 (28)
Version III	4 (27)	1 (30)

Table 4.2: Header (payload) lengths for first and non-first packets for Version I, II, and III. Version III has the least amount of header-overhead for both first and non-first packets.

sends that are matched only by receives specifying the same communication contexts with which the messages were sent.

As can be seen in Figure 4-9(a), as many as 27 payload words can be sent in the first packet. If the MPI message is longer than 27 words, the sending process sends additional packets until the entire message has been sent. Figure 4-9(b) shows the format for non-first packets, which contains a one word header specifying the rank of the sending process. The receiving protocol is such that this one word is enough to associate an incoming non-first packet with an already partially received message. As seen in the figure, a non-first packet can have as many as 30 words of payload. In order to minimize extraneous network traffic, attempts were made to minimize the length of the packet headers. Compared to Version I and Version II, Version III performs better on this front for both first and non-first packets (see Table 4.2 for a summary of header and payload lengths).

Sending to oneself While the general sending algorithm approximates that of Version II quite closely, special provisions had to be made in order to support the MPI-Standard-defined notion of sending to oneself. In other words, the MPI Standard allows one process to issue an MPI_Send to itself, and receive that message using a typical MPI_Recv. To optimize this case, during a send, the sending process checks if the receiver has the same rank and comm as itself. If this is the case, the sender circumvents the process of placing the message on the network and places the message directly into its own Outstanding Messages Structure (described below). Thus, when the matching MPI_Recv is called, the receiver (which is actually the same process that sent the message) finds the message immediately.

Receiving

Version III's receiving design and implementation constitutes the most complex part of Version III's entire design. Like Version II, Version III's receive design is driven by the GDN_AVAIL interrupt; data is only received by the receiving tile when data enters the incoming GDN port for a tile. When the interrupt fires, control flow is transferred to the interrupt handler, which reads as many words as it can from the GDN and places those words in local memory. Similarly to Version II, packets are sorted as they arrive, and individual packets can be received over the course of more than one interrupt handler invocation (as illustrated above in Figure 3-7).

There are a couple of distinct components that work together in an asynchronous fashion to enable any receive action. One component is the interrupt handler that manages the receiving and sorting of incoming packets; this is the most complex part of the receive design, and, as such, will garner the most attention. The other primary receive component is the top-level call that extracts a matching message from the temporary place where the interrupt handler has placed it (in the case when the message has already arrived), or patiently waits until the message does arrive in full. It is, perhaps, already clear that there are some important timing considerations regarding the relative ordering of when a call to MPI_Recv[2] is invoked, and when the message actually arrives. Furthermore, there are cases where, say, only part of the message arrives. For instance, consider the case when, say, only the first two of seven packets arrive, then the call to MPI_Recv occurs, and then the remaining five of seven packets arrive. All of these cases must be handled appropriately to produce a working receive design. Table 4.3 defines the three basic timing cases into which all receives fall.

[2]Other receiving calls may be invoked of course, but in this section MPI_Recv will be used as the canonical example.

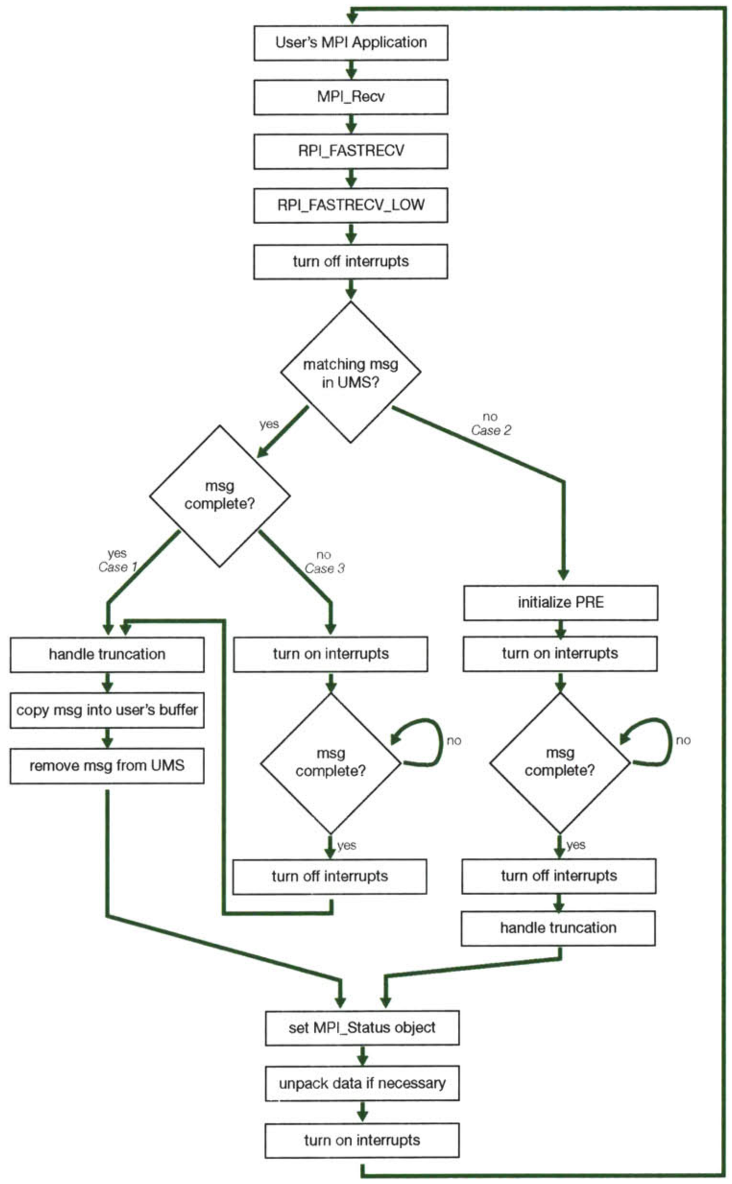


Figure 4-10: Function control flow upon call from user's MPI application to `MPI_Recv`. `MPI_Recv` checks for argument errors and handles trivial cases, and then calls down to `RPI_FASTRECV`, which is implemented in the point-to-point layer. `RPI_FASTRECV` does further error checking on arguments, allocates a temporary buffer if the incoming datatype is not a packed buffer, and then calls `RPI_FASTRECV_LOW`. `RPI_FASTRECV_LOW` checks to see if there is a matching message in the UMS. If not, the PRE (Posted Receive Entry, described below) is initialized, and then it waits until the message has entirely arrived and handles truncation if necessary. If there is a matching message in the UMS (Unexpected Message Structure, described below), but it has not been completely received, it waits until the message has entirely arrived. Once the message has entirely arrived, it then handles truncation, copies the message into the user's buffer, removes the message object from the UMS. Finally, in all cases, the `MPI_Status` object is set and the message is unpacked if necessary. *Note: this flowchart shows nothing regarding the interrupt handler or dealings with the network.*

Case 1	The message is entirely received into a temporary buffer by the interrupt handler before the user code calls `MPI_Recv`.
Case 2	None of the message has been received at the point at which the user code calls `MPI_Recv`.
Case 3	Part, but not all, of the message is received by the interrupt handler before the user code calls `MPI_Recv`.

Table 4.3: Case definitions for relative timings between when the message arrives to a process and when the receiving process actually enacts an `MPI_Recv` call.

Figure 4-10 is a flowchart of the top-level MPI_Recv, starting at the user's MPI application. The actions and tests performed in this figure are completely outside the scope of the interrupt handler. Thus, this figure does not depict any incoming data being read off the network (the interrupt handler, which does read all of the data from the network, is described below). MPI_Recv is similar in function to MPI_Send, shown in Figure 4-4; it checks for argument errors and handles trivial cases, and then calls down to RPI_FASTRECV, which is implemented in the point-to-point layer. RPI_FASTRECV does further error checking on arguments, allocates a temporary buffer if the incoming datatype is not a packed buffer, and then calls RPI_FASTRECV_LOW.

RPI_FASTRECV_LOW and the interrupt handler, drain_gdn, are the two functions that deal at the lowest level of incoming message management. They both interact with global data structures, and ultimately manage the process of moving data from the network into a user's buffer. As mentioned above, drain_gdn, the interrupt handler, is in charge of reading incoming data from the GDN and sorting and assembling packets into messages. The design and implementation of the interrupt handler will be described below. On the other hand, RPI_FASTRECV_LOW is invoked from RPI_FASTRECV, processes calls from MPI_Recv, and is in charge of moving data from global data structures into the user's specified buffer and maintaining correct state in the global data structures.

The two primary global data structures with which RPI_FASTRECV_LOW and drain_gdn interact are the Unexpected Message Structure (UMS) and the Posted Receive Entry (PRE). The UMS is a hash table in which unexpected messages, either partially received or fully received, reside. In other words, when Case 1 arises, and a message arrives before the user calls MPI_Recv, the message is placed into the UMS. The UMS is a robust hash table implementation, keyed upon the message envelope (*i.e.*, the full header from the first packet in a message, seen in Figure 4-9(a)). The UMS hash table stores message ums_message objects as values; the ums_message object has the fields shown in Table 4.4.

The cm_msg_status field specifies the status of the message; it can be inactive, active, but not yet completely received, or active and fully received. Messages are inserted into the UMS immediately after the entire message envelope from the first packet has been completely received. In other words, ums_message objects in the UMS can be either partially received or completely received, and can only exist in the UMS with cm_msg_status fields of STATUS_ACTIVE_DONE or STATUS_ACTIVE_NOT_DONE.

field name	data type	description
`cm_env`	struct lam_ssi_rpi_envl	envelope describing this message
`cm_buf`	data32 *cm_buf	message payload
`cm_dont_delete`	int	specifies whether or not we can delete this message from the cbuf structure
`cm_msg_status`	int	STATUS_INACTIVE, STATUS_ACTIVE_NOT_DONE, STATUS_ACTIVE_DONE

Table 4.4: `ums_message` data structure, used as the value in the UMS.

Low-level receive function: RPI_FASTRECV_LOW Turning attention back to Figure 4-10, it can be seen that `RPI_FASTRECV_LOW` first checks to see if the desired message matches one in the UMS. If so, it checks the `cm_msg_status` to see if the entire message has been received or not. If there is a matching message in the UMS, but it has not been completely received (*i.e.*, STATUS_ACTIVE_NOT_DONE), it is Case 3. In this case, interrupts are turned on and the receiver loops, waiting for the message to become complete. At some point, new data will arrive and the GDN_AVAIL interrupt will fire. The interrupt handler will read in the new data and organize it appropriately. Assuming the user implemented a well-formed MPI application, after one or more interrupt handler invocations, the loop condition in `RPI_FASTRECV_LOW` will eventually become true. Interrupts will then be turned back on in case new data arrives. At this point, the message is complete, so the situation is the same as Case 1. The completed message is first checked for truncation. The MPI Standard specifies that a truncation error has occurred if the user specifies a message length to receive that is shorter than the message that was sent from the sending process. In this case, the `MPI_ERR_TRUNCATE` is returned. Otherwise, data is copied from the `ums_message` structure's temporary buffer into the user's buffer. The message is then removed from the UMS, the `MPI_Status` object is populated appropriately, and data is unpacked if necessary.

The UMS is the data structure into which messages that arrive "unexpectedly" are placed before the user code has requested them. Thus, the UMS handles Cases 1 and 3. On the other hand, the UMS is not used in Case 2, when the receiver posts a request for a particular message before it actually arrives. While this case could be treated somewhat like Case 3, where the receiver just waits until the message has completely arrived, Case 2 presents the opportunity to incorporate a nice optimization. In Case 1 and 3, the unexpected message must be buffered in temporary storage, requiring memory allocations, copying, and deallocations. On the other hand, when the user posts a receive before the

name	data type	description
`env`	struct lam_ssi_rpi_env	copy of desired message envelope
`buf`	void *	pointer to user's target buffer, updated upon copy
`num_bytes_in_sent_msg`	int	number of bytes in message actually sent
`msg_status`	int	STATUS_INACTIVE, STATUS_ACTIVE_NOT_DONE, STATUS_ACTIVE_DONE

Table 4.5: Posted Received Entry (PRE) Fields

message arrives, the target buffer in which the user desires the message to eventually reside is known, removing the requirement for temporary buffer storage. Thus, in Case 2, the UMS is not used; Case 2 employs the Posted Receive Entry.

The Posted Receive Entry (PRE) is the other global data structure with which both the interrupt handler and `RPI_FASTRECV_LOW` interact. The PRE provides a way for `RPI_FASTRECV_LOW` to post a "message" to the interrupt handler, telling it to place a message matching certain characteristics directly into a specified buffer, and not employ the UMS. The PRE can contain at most one entry, as all receives are blocking. Its design can be seen in Table 4.5.

Focusing attention back on Figure 4-10, it can be seen that once `RPI_FASTRECV_LOW` determines that the desired message has no match in the UMS, the PRE is initialized with the appropriate target buffer address and envelope. Then, interrupts are turned on and the receiving process just waits until the `msg_status` field of the PRE becomes true. At some point, the new data will arrive and the GDN_AVAIL interrupt will fire. The interrupt handler will read in the new data and organize it appropriately. After one or more interrupt handler invocations, the loop condition in `RPI_FASTRECV_LOW` will eventually become true, as the message described by the PRE will be fully received. Interrupts will then be turned back on in case new data arrives. At this point, the message data has already been placed in the user's target buffer by the interrupt handler, so no copying must take place. `RPI_FASTRECV_LOW` then handles any truncation errors that may have occurred, sets the `MPI_Status` object appropriately, unpacks data if necessary, and returns.

Interrupt handler: drain_gdn As mentioned above, the interrupt handler, `drain_gdn`, acts as the supplier of messages to `RPI_FASTRECV_LOW`. `drain_gdn` populates and maintains proper state in both the UMS and the PRE while it collects incoming data from the GDN input port. As in

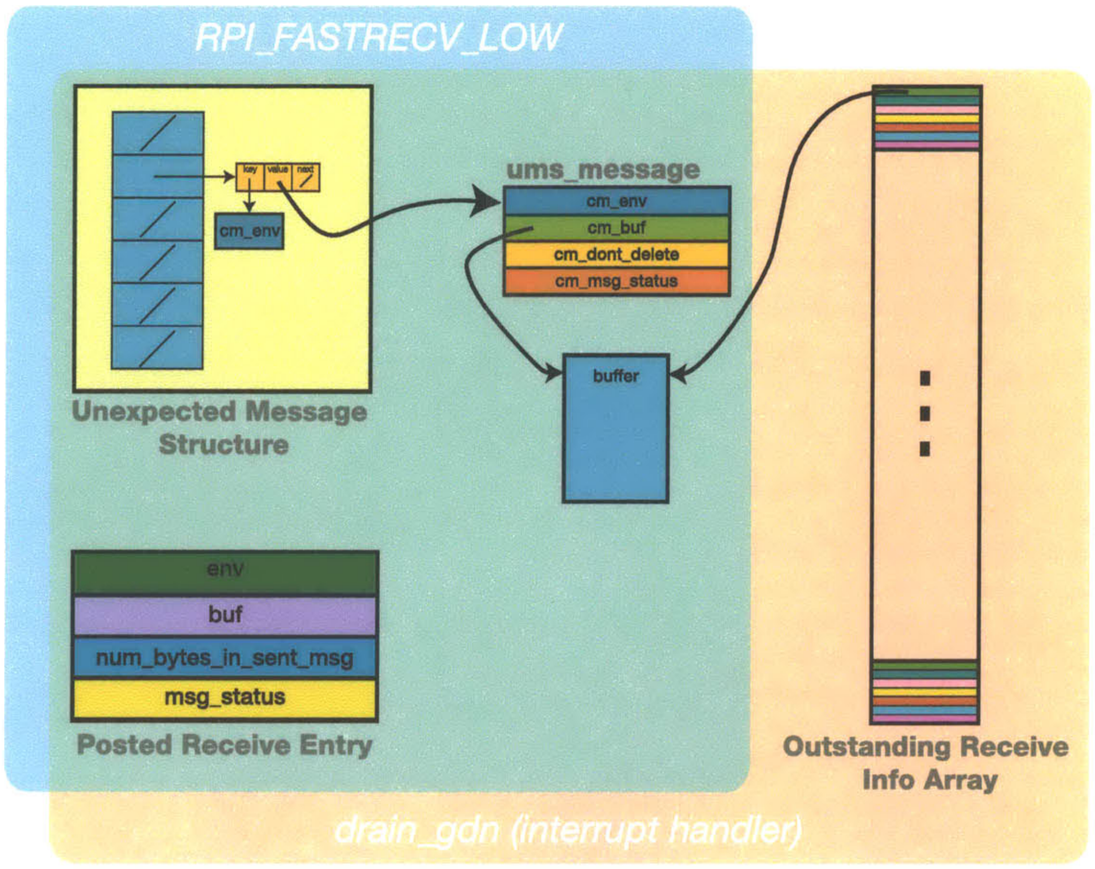


Figure 4-11: The data structures used in performing point-to-point receives. `RPI_FASTRECV_LOW` has access to the UMS (Unexpected Message Structure) and the PRE (Posted Receive Entry), while the interrupt handler, `drain_gdn`, has access to the UMS and PRE, as well as the ORIA (Outstanding Receive Info Array).

Version II, the interrupt handler sorts packets into their respective message by reading their headers, and keeping track of packet and message boundaries. Figure 4-11 depicts the main data structures in the overall point-to-point receive design, and how `RPI_FASTRECV_LOW` and `drain_gdn` interact with them. `RPI_FASTRECV_LOW` interacts with the UMS, the PRE, and the `ums_messages` that reside in the UMS. `drain_gdn` interacts with all of these, as well as the Outstanding Receive Info Array (ORIA). The ORIA is an array of Outstanding Receive Info (ORI) structures, which generally keeps track of bookkeeping information regarding initiated but incomplete receives. In other words, from the time when a particular message starts arriving until the time when it has been completely received, an entry in the ORIA keeps track of important and necessary information that allow the message to be correctly sorted and reassembled. The contents of a ORI can be seen in Table 4.6.

name	**data type**	**description**
`curr_buf`	void *	updated buffer pointer at which to place data
`envl`	struct lam_ssi_rpi_env	copy of associated message envelope
`msg_status`	int	STATUS_INACTIVE, STATUS_ACTIVE_NOT_DONE, STATUS_ACTIVE_DONE
`env_words_outstanding`	int	number of outstanding envelope words
`payload_words_outstanding`	int	number of outstanding payload words
`already_matched`	int	valid values: STATUS_MATCHED | STATUS_NOT_MATCHED, denoting whether this table entry has already been matched (and therefore initialized) by some incoming packet
`PRQ_or_cbuf_status`	int *	pointer to PRQ or cbuf status (depending on whichever is actually holding the data)

Table 4.6: Design of outstanding receive info data structure, which holds information regarding at most one incoming message from a peer process.

The ORIA is indexed by the global rank of the incoming process, based on the understanding that each process can only be sending one message at a time since all sends are blocking. Some fields in an ORI, such as the envelope and number of payload and envelope words outstanding, are self-contained in the ORI (*i.e.,* they are not pointers). On the other hand, as seen in Figure4-11, pointer entries in each ORI can point to either an entry in a `ums_message` or to an entry in the PRE. For instance, the `curr_buf` field and the `PRQ_or_cbuf_status` field are pointers to the relevant fields in a `ums_message` or the PRE, as the message could be either Case 1/3 or Case 2. Thus, the ORI structures were designed to be sufficiently general to accommodate all incoming messages, whether they were unexpected or not.

As shown in Figure 3-7, packets can be received over the course of multiple invocations of the interrupt handler. To this end, a static variable named `curr_sender_rank` was used to keep track of the global rank of the currently incoming packet. This variable is first set to the global rank of the first incoming packet of the program, and is updated upon the completion of a packet and the start

name	data type	description
curr_sender_rank	int	retains the rank of the current sender. Its value is -1 when there is no outstanding receiver.
words_recvd	int	retains the total number words that have been received thus far in the currently incoming packet.

Table 4.7: Static variables in the interrupt handler

of a new packet. Again, Figure 3-7 illustrates how such packet interleaving can occur.

Additionally, another static variable was needed to keep track of how many words have been received thus far on the currently incoming packet. This variable, named words_recvd, is updated after either a packet boundary has been reached, or when the interrupt handler has no more words to receive (*i.e.*, the GDN input queue is empty). words_recvd's most important function is to assist in determining packet boundaries, and is used extensively throughout the interrupt handler's implementation. The static variables used in the interrupt handler are summarized in Table 4.7.

Now that the data structures and important variables have been presented, attention is now turned to the interrupt handler's control flow. Figure 4-12 shows the control flow of drain_gdn. Upon each call to the function, execution begins at the START node. The first test checks if the last time the handler was called it returned before the entire packet arrived. If this condition is false, as it is in the beginning of the program, the hander receives one word from the GDN. Note that the handler is guaranteed that there is at least one word available on the GDN input port, as it is only called when incoming data is available. However, the check to ensure data is available on the network exists because this node in the control flow can be reached in other ways.

The first word in any packet is the global rank of the sending process. This value is used to index into the ORIA, which holds information regarding a potentially active outstanding message from the sender. At the beginning of the program, the entry ORIA[grank][3] is in the inactive state. Thus, the check to see if the entry is active will return false. Thus, ORIA[grank] is initialized, and the handler will attempt to read as many envelope words as it can. If, perhaps due to network congestion, the envelope is not received completely, the ORIA[grank] entry is updated to reflect how many words of the envelope have been received, curr_sender_rank is updated to be the just-received grank, and the handler returns. On the other hand, if the entire envelope is received, the handler then checks if there is already an entry in the ORIA that already matches, which is never true in the case

[3]The notation ORIA[grank] refers to the ORI object that exists in the ORIA associated with the global rank grank.

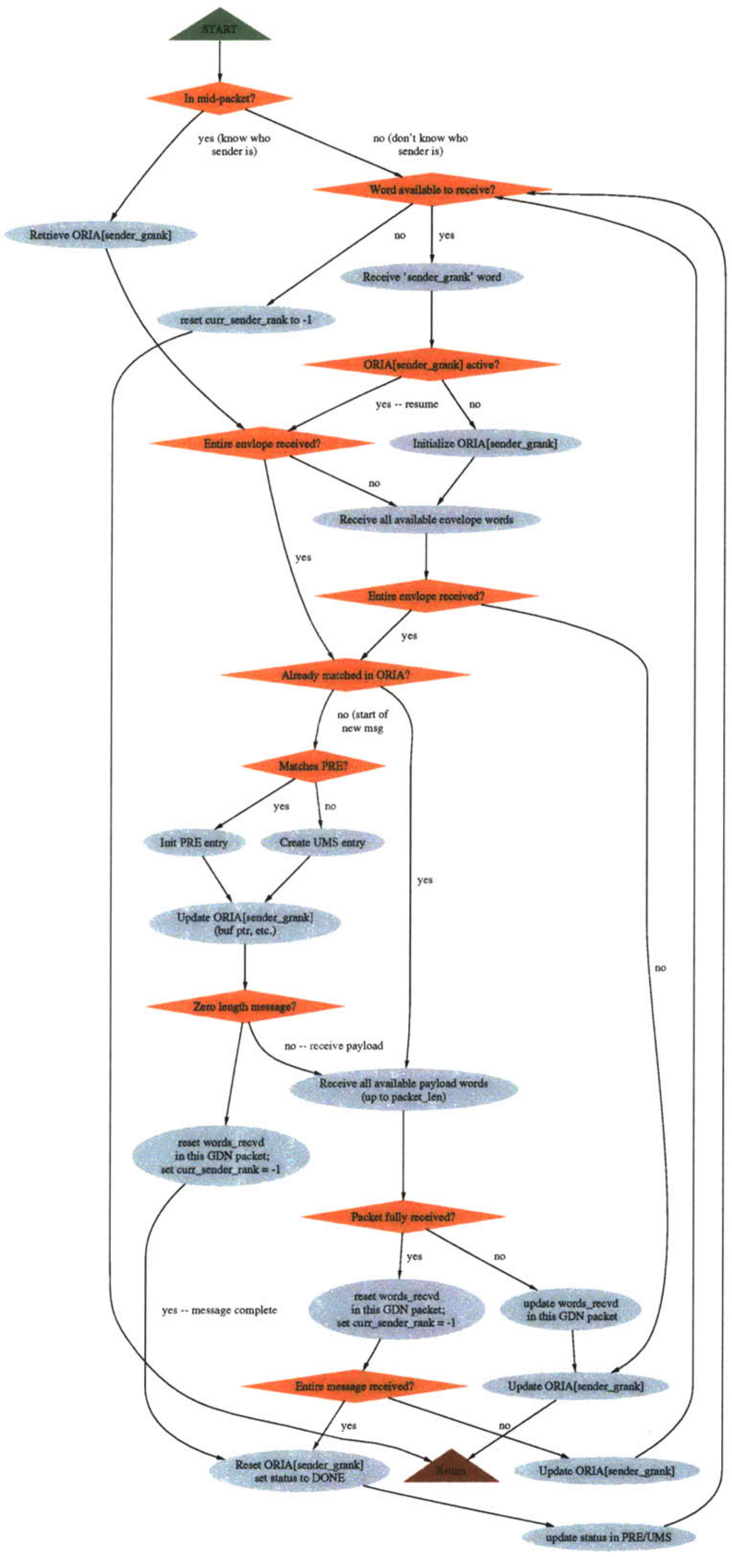

START
In mid-packet?
yes (know who sender is)
no (don't know who sender is)
Word available to receive?
Retrieve ORIA[sender_grank]
no
yes
Receive 'sender_grank' word
reset curr_sender_rank to -1
ORIA[sender_grank] active?
yes -- resume
no
Entire envlope received?
Initialize ORIA[sender_grank]
no
Receive all available envelope words
yes
Entire envlope received?
yes
Already matched in ORIA?
no (start of new msg
Matches PRE?
yes
no
Init PRE entry
Create UMS entry
yes
Update ORIA[sender_grank] (buf ptr, etc.)
no
Zero length message?
no -- receive payload
Receive all available payload words (up to packet_len)
reset words_recvd in this GDN packet; set curr_sender_rank = -1
Packet fully received?
yes
no
yes -- message complete
reset words_recvd in this GDN packet; set curr_sender_rank = -1
update words_recvd in this GDN packet
Entire message received?
Update ORIA[sender_grank]
yes
no
Reset ORIA[sender_grank] set status to DONE
Return
Update ORIA[sender_grank]
update status in PRE/UMS

Figure 4-12: `drain_gdn`'s control flow graph.

of a new message[4]. The "no" branch of the "Already matched in ORIA" node is taken. At this point, the handler checks if the PRE contains a message request that matches the newly received envelope. If so, the PRE is initialized and the new incoming data is read directly into the buffer specified by the PRE (*i.e.,* the user's target buffer). On the other hand, if there is no match in the PRE, a new `ums_message` is created and initialized. The new incoming data will be temporarily buffered in this new `ums_message` object. In both cases, ORIA[grank] is updated so that the `curr_buf` field and `PRQ_or_cbuf_status` is set appropriately (pointing either to the `ums_message` or the PRE, depending on the case).

Next, the message envelope size field is checked for the case of a zero-length message, a perfectly valid notion according to the MPI Standard. If this is the case, `words_recvd` is reset to 0, and `curr_sender_rank` is reset to -1. Furthermore, ORIA[grank] is reset, and the status field pointer is traversed and set to STATUS_ACTIVE_DONE. Finally, the handler checks to see if more data has arrived on the GDN input queue, thereby starting the entire process again.

If the message has a non-zero length, the handler attempts to receive as many words of payload as it can, up to the length of the packet. If the packet was fully received in this attempt, `words_recvd` is reset to 0 and `curr_sender_rank` is reset to -1. If the receiving of this packet also constituted the receiving of the entire message, ORIA[grank] is reset, the status in the `ums_message` or PRE is set to STATUS_ACTIVE_DONE, and the GDN input queue is checked for more data, in which case the entire receiving and sorting process recommences.

If, after receiving as many payload words as it can, the handler finds that it finished receiving a packet but that packet did not constitute the last packet in the message, it updates ORIA[grank] and checks to see if more data is available to be received, in which case the entire process begins again. If, on the other hand, the hander is unable to receive an entire packet, perhaps due to congestion in the network, `words_recvd` and ORIA[grank] are updated and the handler returns.

Finally, in the cases when the interrupt handler is called and it discovers that the last invocation of the handler must have returned with an only partially received packet, it does its best to finish receiving the packet this time. First, it checks in the ORIA[grank] to see if the envelope has been entirely received. If so, it proceeds to check if the message has already been started or not, and proceeds as shown in the figure. If the entire envelope has not yet been entirely received, it attempts to receive the entire envelope, and proceed normally.

The interrupt handler design is quite complex due to the fact that packets can be interleaved,

[4]While this check may seem redundant in the figure, the code is optimized to avoid this case.

received over multiple invocations of the interrupt handler, and because messages can either be unexpected or requested. However, the data structures and clean interface and protocols aptly manage this complexity, resulting in a robust and efficient design.

Special Compiler Directive In both "msg complete?" loops shown in Figure 4-10, special compiler directives had to be added to make sure the compiler did not "optimize away" the load that read the status flags. rMPI was compiled with the `-O3` flag, which optimized away the load, assuming there was no way it could change. Of course, in this implementation, the interrupt handler *can* change the value. The necessary code fix can be seen in Figure 4-13.

Message Matching Conditions In a number of cases above, the notion of incoming messages "matching" a user's request arises. As message is considered "matching" if the context ids match, the message sources match, and the message tags match. Additionally, the requesting envelope may contain source and/or tag wildcards (*i.e.*, `MPI_ANY_SOURCE` and/or `MPI_ANY_TAG`). The implementation of the envelope comparison function can be seen in Figure 4-14.

4.4 Implementation and Development

rMPI was developed both using the Raw infrastructure (*e.g.*, simulator and associated tools) as well as standard unix utilities on a Linux workstation. The sheer size of the library severely slowed the development cycle using the Raw tools – compilation and simulation of a reasonable program typically took tens of minutes. Because of this, the portions of library that are independent of Raw were developed using the comparatively fast unix tools. For instance, Raw-specific functions were temporarily "stubbed out" to allow compilation with the standard gnu C compiler (gcc), and execution natively on the x86 platform. Furthermore, other unix utilities such as valgrind [8] and gdb [1] were quite helpful in isolating memory leaks and other difficult-to-find errors in this stubbed out version. As discussed further in Chapter 5, this process was also used in testing the higher level functionality of the library. Of course, much of the library relies on functionality only available on Raw, and therefore had to be run on the Raw simulator or the Raw hardware. Overall, the hybrid development approach sped up development time considerably.

4.5 Limitations

While rMPI (Version III) supports a large portion of the MPI standard, it does not support some key classes of functions. One of the most fundamental limitations of rMPI is that it does not support non-blocking sends and receives (and higher-level calls that use the lower-level non-blocking send and receive calls). The decision was made to not include non-blocking calls in rMPI largely because of Raw's inherently single-threaded nature. Implementing non-blocking calls would not only be a significant challenge, but would probably suffer a large enough performance loss that users would most likely not benefit from the merits of the non-blocking primitives. Nevertheless, the implementation of a robust and high-performance non-blocking send and receive primitives is a potential area of future work for rMPI.

rMPI also does not support sub-word datatypes, such as `MPI_BYTE`. Thus, to send a single byte, or an array of individual bytes (*e.g.*, a string of characters), the rMPI user would have to pack the data into a word-aligned array, and send that. Correspondingly, the receiver would have to do any necessary unpacking, possibly by ignoring head-prepended or tail-postpended bytes.

Additionally, most of the MPI-2 standard is not supported in rMPI. For example, rMPI does not support process creation and management (such would be quite difficult on Raw, a single-threaded platform), one-sided communication, and I/O. These functions are better served directly by Raw's infrastructure.

Finally, rMPI imposes some limitations on the point-to-point communication *modes*. The mode allows the user to specify the semantics of a send operation, which effectively influences the underlying data transfer protocol. The MPI standard defines four sending modes: *standard*, *buffered*, *synchronous*, and *ready*. A *standard* mode send does not necessarily mean that a matching receive has started, and no guarantee can be made on how much buffering is done by MPI. In *buffered* mode, the user must explicitly provide buffer space, which can be used by MPI if buffering is necessary. In *synchronous* mode, the sender and receiver use rendezvous semantics. Finally, in *ready* mode, the sender asserts that the receiver has already posted a matching receive. This mode allows the underlying transport protocol to take advantage of extra knowledge from the application, potentially increasing efficiency and performance. rMPI effectively implements standard mode sends, and leaves the remaining send modes as future work.

```
prq_msg_status_addr = &(lam_ssi_rpi_raw_posted_recv_entry.msg_status);
// force a load into flag every iteration. gcc was optimizing this away
__asm__ volatile("lw %0, 0(%1)"    : "=r" (flag)    : "r" (pre_msg_status_addr));

// spin loop until entire msg has been received.
while (flag != STATUS_ACTIVE_DONE)
{
        __asm__ volatile("lw %0, 0(%1)"    : "=r" (flag)    : "r" (pre_msg_status_addr));
        continue;
}
```

Figure 4-13: Special compiler directive required to prevent compiler from optimizing repeated load away. The value of this load can change because the interrupt handler changes it when the status of a message in the UMS or the status of the PRE changes.

```
int envelope_compare(struct envl *p1, struct envl *p2)
{
        if ((p1->ce_cid == p2->ce_cid) &&
                ((p1->ce_rank == p2->ce_rank) || (p2->ce_rank == MPI_ANY_SOURCE)) &&
                ((p1->ce_tag == p2->ce_tag) || (p2->ce_tag == MPI_ANY_TAG)))
                return(0);

        return(1);
}
```

Figure 4-14: Envelope comparison function, which determines whether two envelopes can describe the same message. Envelope p2 may contain wildcards, but envelope p1 may not.

Chapter 5

Experimental Evaluation and Analysis

This chapter presents experimental results and analysis that show rMPI Version III provides a scalable, deadlock-free, and low-overhead MPI implementation for the Raw architecture. It discusses various performance metrics on a variety of kernel benchmarks and applications.

Section 5.1 discusses the testing methodology used to verify the correctness and standards-compliance of rMPI. After a description of the evaluation methodology and test-bed in Section 5.2, the experimental results are presented. The results include evaluation of the overhead incurred by rMPI, a number of standard MPI benchmark kernels for varying message sizes, and some MPI applications.

5.1 Testing and Validation

One of the goals of the rMPI testing effort was to ensure that rMPI upheld the interface and semantics of the MPI standard. Given the vastness of the MPI standard, testing rMPI was a significant task. To facilitate and expedite development and testing, rMPI was first implemented on a Linux workstation by stubbing out Raw-specific calls. For instance, a read of the GDN was simply stubbed out as a function that returned an integer. In some cases, the returned value was from a pre-initialized array that contained data mimicking real MPI packet data, complete with headers and payload. In other words, using the names employed in Figure 4-3, the High-Level MPI Layer, Collective Communications Layer, and parts of the Point-to-Point Layer were tested before the Raw processor was brought into the picture. This enabled large portions of the code base to be tested using powerful Linux tools such as Valgrind [8] and gdb [1] that are not available on Raw. Furthermore, the code-compile-run-debug cycle on Linux is significantly faster than that on Raw, as the Raw compiler and simulator are quite slow for such large programs compared to their Linux analogs. Finally, this software development method constrained the space of potential errors by removing all potentially unreliable hardware and software mechanisms on Raw.

After rMPI was tested on Linux as much as possible, the development effort was moved to the Raw platform. A test consisting of a simple send-receive pair, which contained a single one-word message, was first developed, and low-level errors were removed until the test passed. The success of this particular test, albeit trivial, ensured that many components of the system were functional, and was therefore a significant milestone. The simple send-receive pair test was extended to longer messages, and then to multiple senders and receivers. Further testing ensured that receive cases 1, 2, and 3 (defined in Table 4.3) were all functional.

After a number of custom tests validated basic operation, the Intel MPI Benchmark (IMB) Suite was employed to verify more sophisticated usage. The IMB is a well-known and highly-used test suite that is reminiscent of the MPICH and IBM test suites. Its objectives[1] are:

- Provide a concise set of benchmarks targeted at measuring the most important MPI functions,
- Set forth a precise benchmark methodology,
- Don't impose much of an interpretation on the measured results: report bare timings instead.

The included tests stressed the collective operations (*e.g.*, `MPI_Allgather`, `MPI_Scatter`, etc.), datatype management, use of wildcards, communication context management, and so on. Overall, these tests stressed the system in such a way that a number of bugs were determined and subsequently fixed. Throughout development, rMPI was tested with regression tests, mostly consisting of tests from the IMB suite.

Tests were run on both the Raw simulator and hardware, both of which had their pros and cons as a testing platform. The simulator enabled careful debugging, network visualization, and, probably most importantly, deterministic timing. On the other hand, the hardware enabled significantly faster processing speeds, but allowed timing inconsistencies. Since the size of executables required to run rMPI programs is so large (due to the large size of the library itself), running programs on the hardware also required the use of the software instruction caching system. The software instruction caching system, which is quite complex, contained a small number of errors dealing with the `GDN_AVAIL` interrupt that had to be fixed before rMPI could be run on the hardware. On the other hand, software instruction caching was not necessary when running on btl, as the simulator allowed

[1] Objectives extracted from the IMB README file.

the size of the instruction cache to be changed. Overall, a combination of both the hardware and the software platforms was used to thoroughly test rMPI.

5.2 Experimental Methodology

This section describes the infrastructure and framework used to evaluate various aspects rMPI. While rMPI was tested and verified on both the hardware and the software simulator, the results presented in this section were collected from evaluations that were run on the simulator. The simulator was chosen as an evaluation platform because it allowed the memory system to be modified such that its performance did not dominate the overall performance of the benchmarks being run. Many MPI programs have an extremely large data footprint, with message sizes ranging from a few words up to many megabytes. In order to evaluate the characteristics of rMPI, and not Raw's memory system, the size of the data cache was set to 1MB. Given the large size of the library itself, rMPI requires software instruction caching in order for the object file to fit in memory. In order to focus the evaluation on rMPI and not the software instruction caching system, the instruction cache size was also increased to 256KB, allowing both the rMPI library and the user program to fit into the on-chip instruction cache. These values were upheld for all of the simulations, including the serial versions of the benchmarks.

To access how well rMPI met its goals, three classes of benchmarks were run. First, an overhead evaluation was performed relative to the native GDN for varying sizes of messages. Next, a number of kernel benchmarks were run to assess the scaling properties of various kernel operations that are often used in large MPI applications. Finally, a number of real applications were run to determine the speedup that was achieved with increasing parallelism.

In all of the tests, the caches were "warmed up" by running some appropriate subset of the test before timing was started. This turned out to be quite necessary, as experimentation showed that cold cache misses account for many cycles near the start of each program.

In evaluations that varied the number of participating tiles, a new communication context was constructed using `MPI_Comm_split` with the desired number of tiles. Given the nature of `MPI_Comm_split`, tests with N tiles used tile numbers 0 to $N-1$. This configuration crippled some of the results because of the small amount of buffering in the network. For instance, a four tile configuration using tiles 0, 1, 4, and 5 (see Figure 5-1) would have most likely yielded better results than the configuration using tiles 0, 1, 2, and 3, as was used in the evaluations. However, the choice was made to evaluate rMPI using programs written in the way that MPI developers would be most likely to write programs, and

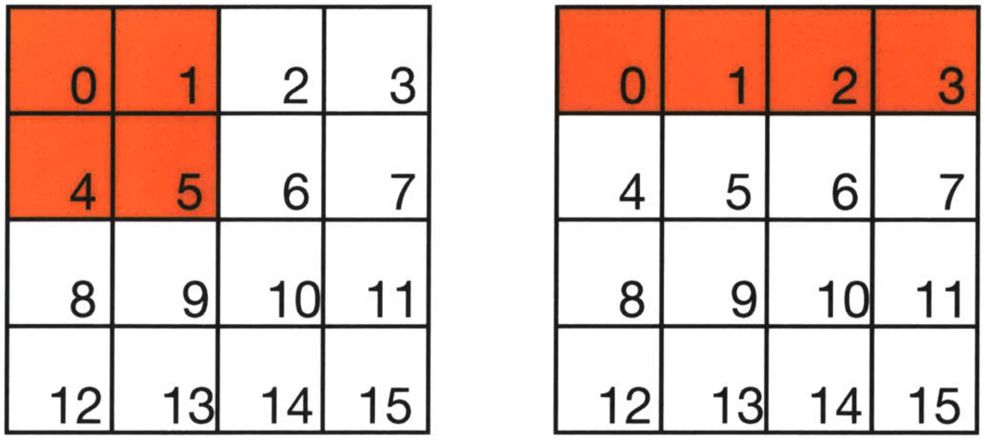


Figure 5-1: Two possible 4-tile configurations. The configuration on the left shows tiles 0, 1, 4, and 5 participating in the computation, while the configuration on the right shows tiles 0, 1, 2, and 3 participation in the computation. The right configuration was used for the evaluations, as it is most likely the manner in which most programmers would develop their programs.

number of tiles in evaluation	**root tile**
2	1
4	2
8	6
16	6

Table 5.1: Locations of the root tile in collective operations for different numbers of tiles

thus the straightforward approach was employed.

For tests that used collective operations containing a root (*e.g.,* broadcast, reduce, gather), the root was placed somewhere near the spatial center of the group of tiles executing the program. Table 5.1 lists the specific locations of the root for different numbers of tiles. Tiles on the right side of the center of the chip were preferred over those on the left, as those tiles are closer to off-chip DRAM, and thus have slightly lower cache miss latency.

5.3 Overhead Evaluation

One of the primary goals of rMPI as defined in Chapter 3 is that it be low overhead. Regardless of how easy it may be to write programs using the MPI interface provided by rMPI, application developers would probably not use rMPI if the overhead was prohibitively high. It turns out that the optimizations employed in Version III of rMPI yielded pleasing results. An evaluation comparing the

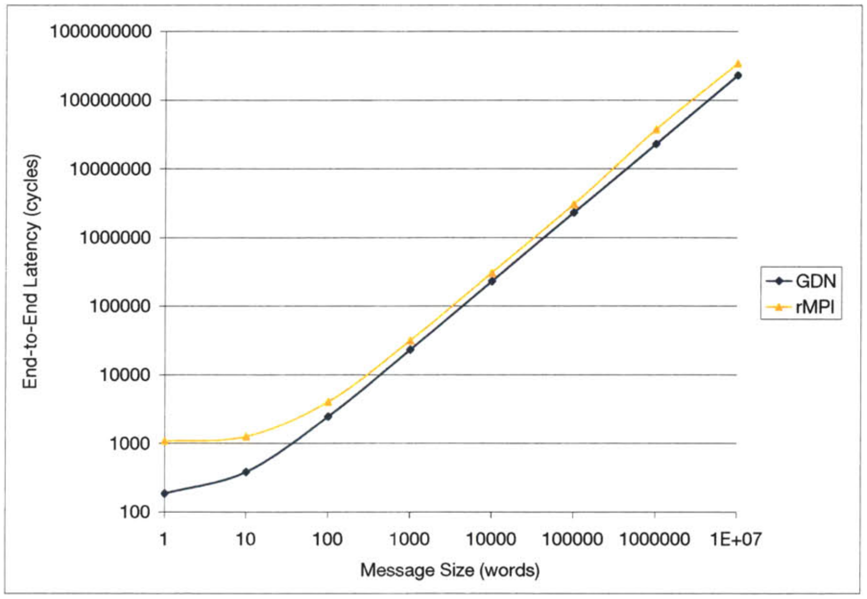


Figure 5-2: Overhead analysis of rMPI. Messages of varying sizes were sent from tile 3 to tile 7 using both the GDN and rMPI. The overhead of rMPI is seen to be minimal, and its scaling properties closely resemble that of the GDN.

end-to-end latency of messages transmitted from tile 3 to tile 7 was run using both rMPI and the GDN natively, with message sizes increasing from 1 word to 10 million words (about 40MB). Figure 5-2 shows the resulting latencies. As expected, rMPI imposes a constant overhead that is especially noticeable at small message sizes. This overhead can be attributed to the extra data structures and the large header that rMPI places at the start of any MPI message.

As the message size grows from 10 to 100 words, however, the difference of end-to-end latencies between both transport mechanisms narrows. Furthermore, both mechanisms scale almost identically. The almost-constant difference between the GDN graph and the rMPI graph can be attributed to two overhead components. First, 1 out of every 31 words, or 3.2% of words transmitted, are rMPI packet headers (save for the first packet, which is negligible for large messages). Thus, 96.8% of words transmitted using rMPI are actual payload words, compared to 100% of words transmitted using the GDN natively. Second, the extra computation and memory accesses necessary for data structure management. Overall, this study shows that the overhead imposed by rMPI is quite low, and there-

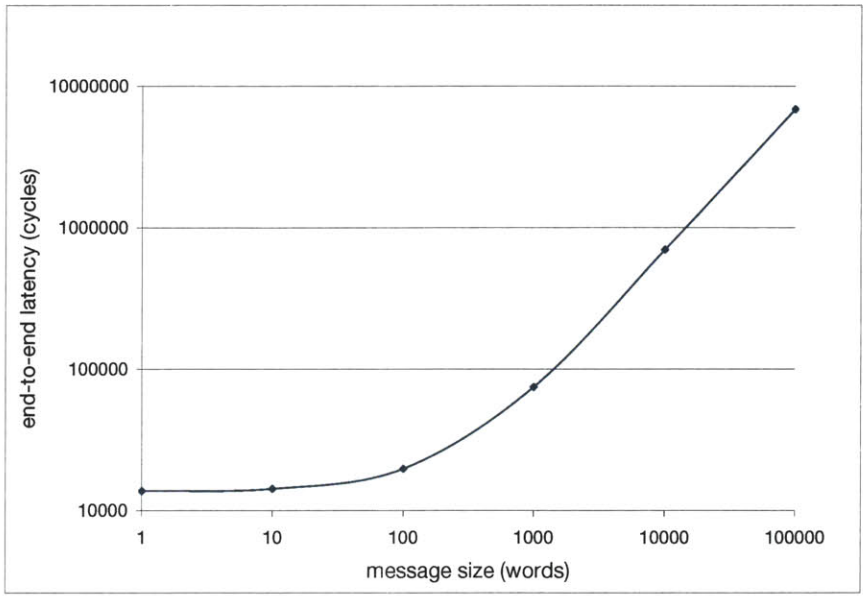


Figure 5-3: End-to-end latency of Ping-Pong benchmark, with message sizes increasing from 1 word to 100,000 words.

fore using rMPI does not cause performance to be significantly sacrificed.

5.4 Kernel Benchmarks

This section presents the results of kernel benchmarks that evaluate the scaling properties of rMPI as message lengths are increased. The kernels presented in this section were extracted from the Intel MPI Benchmark (IMB) suite, a set of commonly-used kernels for benchmarking MPI implementations and extensions. In all kernels presented in this section, the end-to-end latency was measured with message sizes increasing from 1 word up to 100,000 words. Note that end-to-end latency refers to the number of cycles from just before the first send is enacted to just after the last receive is completed.

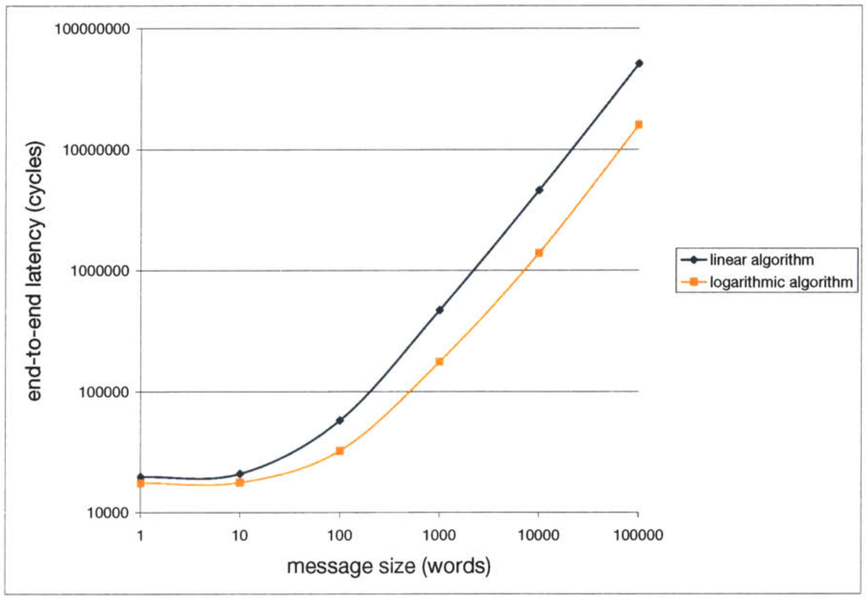


Figure 5-4: End-to-end latency of broadcast benchmark, with message sizes increasing from 1 word to 100,000 words.

5.4.1 Ping-Pong

Ping-pong is a classic MPI benchmark involving two processes: process A sends a message to process B, and then process B bounces the message back to process A. Overall, each process executes one send and one receive. The end-to-end latency was measured, and can be seen in Figure 5-3. As the message size increases, the latency stays somewhat constant up until a message size of 100 words. The graph shows an elbow at 100 words, above which the latency increases quadratically. This change can be attributed to the increased congestion on the GDN and increased contention in the memory network due to larger message sizes. This phenomenon was also evident on the native GDN, as seen in the overhead evaluation in Section 5.3, and should not be significantly attributed to rMPI itself.

5.4.2 Broadcast

The broadcast benchmark was also taken from the IMB suite. It consists of a root node (process 6, in this case) broadcasting messages of various lengths to the 15 other processes. Figure 5-4 shows the

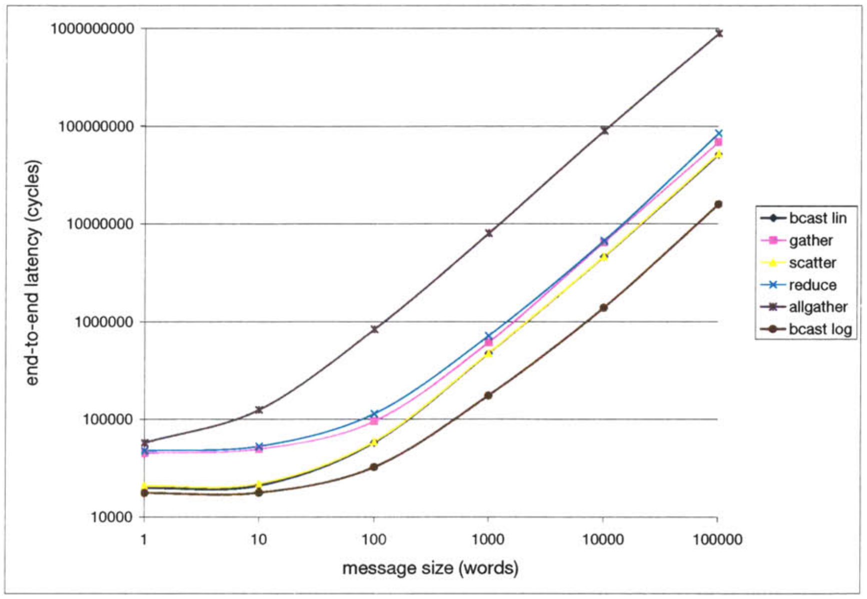


Figure 5-5: End-to-end latency of gather, scatter, reduce, and allgather benchmarks, with message sizes increasing from 1 word to 100,000 words.

end-to-end latencies from the time the broadcast was initiated by the root, until the time when all receiving processes have received their message for both linear and logarithmic broadcast algorithms. As depicted in Figure 3-16, collective communication operations can be implemented in a variety of algorithms. Figure 5-4 shows that especially for large messages, the logarithmic approach achieves a 3.2x speedup over the simplistic linear approach. In both cases, there exists an elbow in the graph in between 10 words and 100 words; this can be attributed to the same reasons described previously in Sections 5.3 and 5.4.1.

5.4.3 Gather, Scatter, Reduce, and Allgather

A number of other benchmarks were run to access the performance and scaling properties of other commonly used MPI primitives. These benchmarks were, once again, taken from the IMB suite, and run with an increasing message length. Figure 5-5 shows the resulting end-to-end latencies for all of the benchmarks. The best-performing benchmarks are the logarithmic version of broadcast, the linear version of broadcast, and scatter. This makes sense because in all of these cases, messages emanate from a root node and all non-root processes receive just one message. Reduce and gather take slightly longer than broadcast and scatter, which is to be expected, as many messages converge on one process, prompting network congestion and significant memory traffic to and from the root node. Finally, the allgather benchmark clearly took significantly longer than the other benchmarks, but this is largely to be expected since allgather is simply a gather operation followed by a broadcast operation. In some sense, the curve for allgather is simply the sum of the gather curve and the broadcast curve, modulo savings due to communication overlap and losses due to increased congestion.

Overall, Figure 5-5 shows that all of the collective operations that were evaluated exhibit similar scaling properties that mimic the general scaling characteristics of the underlying communication medium, the GDN.

5.5 Applications

This section presents four real MPI applications. The applications are evaluated and speedup results are reported for a varying number of tiles. The source code for the applications was originally written by the author, but was inspired from applications in the LAM/MPI test suite, [21], and [22].

5.5.1 Integration with the Trapezoidal Rule

The definite integral of a nonnegative function $f(x)$ can be computed using the trapezoidal rule. Recall that the definite integral from a to b of a nonnegative function $f(x)$ can be viewed as the area bounded by the function $f(x)$, the x-axis, and the lines $x = a$ and $x = b$. To estimate this area, the area under the curve can be filled in with n trapezoids, where n is a parameter that can be adjusted depending on the desired fidelity of the result. The width of each trapezoid is given by $h = (b-a)/n$, the area of the trapezoid is given by

$$A_i = \frac{1}{2}h[f(x_{i-1} + f(x_i)],$$

and the area of the entire approximation will be the sum of the areas of all the trapezoids. This result is given by

$$A = \sum_{i=0}^{n} A_i = [f(x_0)/2 + f(x_n)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1})]h.$$

$$\begin{aligned}\int_0^{1048576} & x^2 * 9 * x + (8 - 2 * x^3)/(x/8 * x + 838387272/2.18 * x) + \ldots \\ x^2 * 9.19191 * x + (8/(x^3 * 3717.121209) * x^2 - 2 * x^2)/x^2 * 9 * x + (8 - 2/x)/\ldots \\ (x + 8 + x + 838/387272/38388221/x) + \ldots \\ (x^2) * 9.19191 - x + (8/(x^3 * 3717.121209) * x^2 - 2 * x^3)dx\end{aligned}$$

Figure 5-6: The integral evaluated during the trapezoidal rule integraion application.

This application was parallelized using MPI primitives by allocating different portions of the integration to different processes, and later combining the results. The somewhat complicated polynomial seen in Figure 5-6 was integrated for this evaluation with $n = 16384$. The speedups, which can be seen in Figure 5-7, are nearly linear. The speedup of adding a second tile is not great due to the increased overhead of communication, but 4, 8, and 16 tiles achieve substantial speedups. The speedup for 16 tiles is 77% of perfect speedup, compared to 98% and 94% of perfect speedup for 4 and 8 tiles, respectively. This relative slowdown is most likely attributable to heavy network congestion when 15 tiles must coalesce their partial results together.

5.5.2 Pi Estimation

The pi (π) estimation application uses an iterative method to estimate the value of the mathematical constant pi, and is similar to the integration via trapezoidal rule application. Pi is computed by integrating the function seen in Figure 5-9; since this integral is much simpler and the approximation algorithm is slightly different, the speedup results look similar, but not exactly the same as those for the trapezoidal rule application. The speedup results for this application can be seen in Figure 5-8. Not surprisingly, as the communication overhead is low, the resulting speedups are quite high for 8 and 16 tiles, at 95% and 91% of perfect speedup, respectively. The relatively low performance with 2 and 4 tiles is most likely due to the extra overhead imposed by the rMPI library.

5.5.3 Jacobi Relaxation

The Jacobi Relaxation method is an iterative algorithm that can solve linear systems of the form $Ax = b$, where the diagonal elements of A are nonzero. This method begins with an initial approximation x^0 to the solution vector, and repeatedly computes a new approximation x^{k+1} from the current approximation x^k. x_i^{k+1}, the ith element of x, is iteratively computed using the following

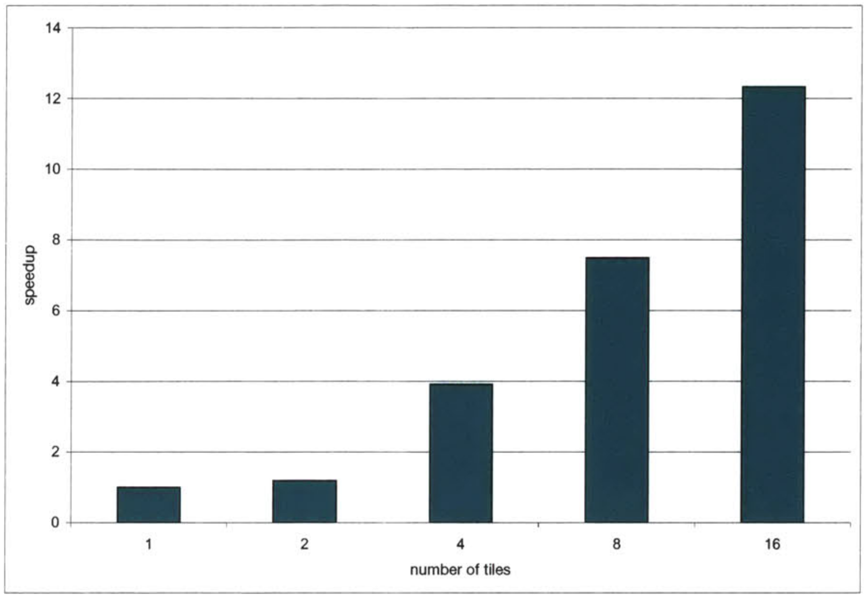


Figure 5-7: Speedup of integration with trapezoidal rule application for 1, 2, 4, 8, and 16 tiles.

formula:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^k \right)$$

The algorithm was parallelized by distributing data with a row-wise block-striped decomposition. In other words, each process manages an $(n/p) \times n$ region, where the 2-dimensional array is of size $n \times n$ and there are p processes partaking in the computation. This algorithm exhibits high locality, as the only data that must be communicated are the top and bottom row of each processes' block of data upon each iteration. Additionally, this application would achieve higher performance with a different data distribution scheme (block-oriented instead of row-oriented).

Figure 5-10 shows the resulting speedup when jacobi relaxation was applied to a 3200×3200 2-dimensional matrix. The speedup for two tiles was marginal: only a 20% gain was realized. However, the speedup for four tiles increased to 2.3x, the speedup for eight tiles increased to 4.4x, and the speedup for sixteen tiles increased to 9.3x. Given the large data set and the amount of communication required to carry out this computation, the resulting speedup is 9x for sixteen tiles.

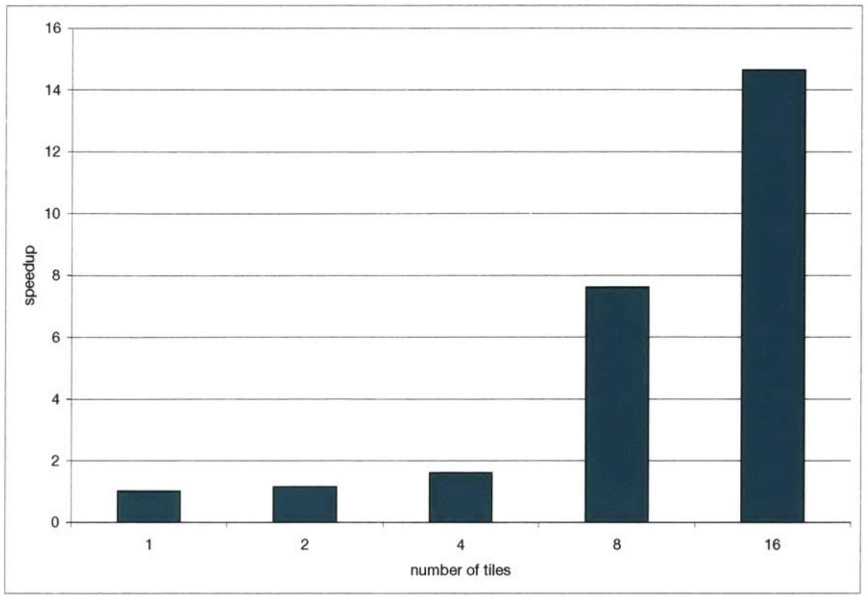


Figure 5-8: Speedup of pi estimation application for 1, 2, 4, 8, and 16 tiles

$$\int_0^1 \frac{4}{(1+x^2)}$$

Figure 5-9: The integral evaluted during the pi estimation application.

5.5.4 Matrix Multiplication

The product of an $l \times m$ matrix A and an $m \times n$ matrix B is an $l \times n$ matrix C whose elements are defined by

$$c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} b_{k,j}$$

The straightforward sequential algorithm requires $l \cdot m \cdot n$ additions as well as $l \cdot m \cdot n$ multiplication. Thus, multiplying two $n \times n$ matricies yields a time complexity of $\Theta(n^3)$. This operation can be parallelized in many ways, some more straightforward than others. For this evaluation, the matrix operation $AB = C$ was parallelized using a row-oriented algorithm, whereby each of the p processes is responsible for computing l/p rows of C. Each process, therefore, must have access to l/p rows of

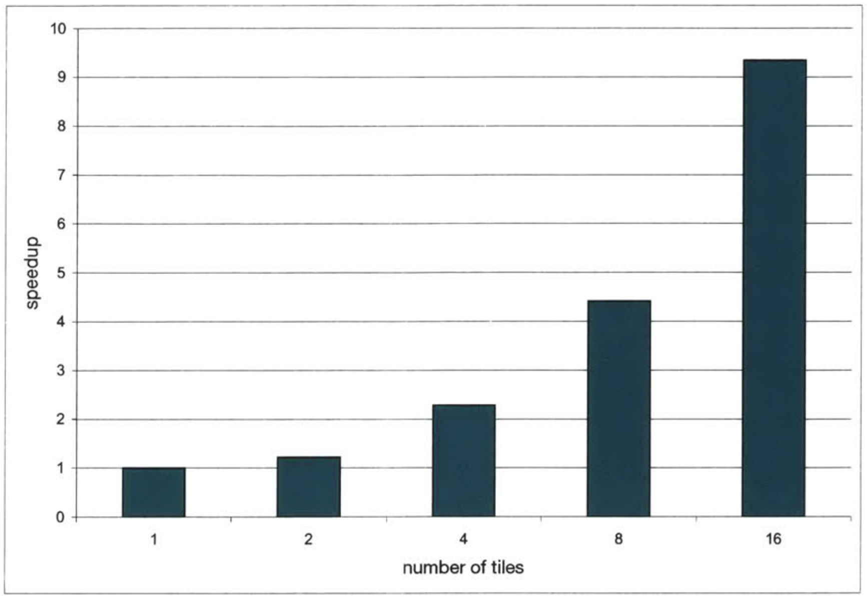


Figure 5-10: Speedup of Jacobi Relaxation on a 3200 × 3200 2-d matrix for 1, 2, 4, 8, and 16 tiles

A and all of B. Figure 5-11 shows how the matricies were partitioned. For this evaluation, matrix A had dimensions of 640 × 150, matrix B had dimensions of 150 × 700, and matrix C had dimensions of 640 × 700; all values were double floating point numbers.

Figure 5-12 shows the resulting speedup results. The running time for the parallelized algorithm on two tiles was longer than that for just one tile. This is not terribly surprising, as the cost of communicating a large chunk of data overwhelms the savings in computation. However, four tiles yields a speedup of just over 2.8x, eight tiles yields a speedup of 5.8x, and sixteen tiles yields a speedup of 8.5x. Clearly, the large amount of data that must be communicated prevents perfect speedup, but the resulting performance is nonetheless quite strong.

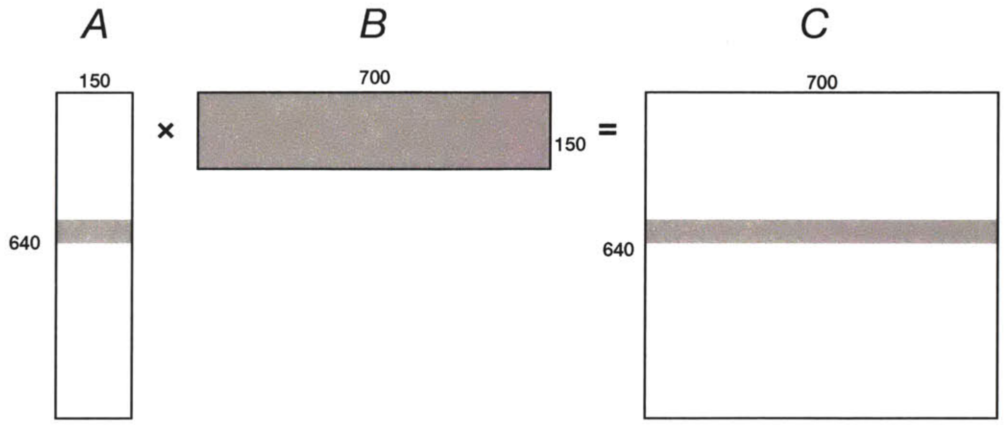


Figure 5-11: Row-oriented parallel matrix multiplication algorithm, showing actual dimensions of matricies used in evaluation. Each process is responsible for computing n/p rows of C, where n here is 640, and $p \in \{1, 2, 4, 8, 16\}$. The shaded regions shows the data referenced by a particular process p_6.

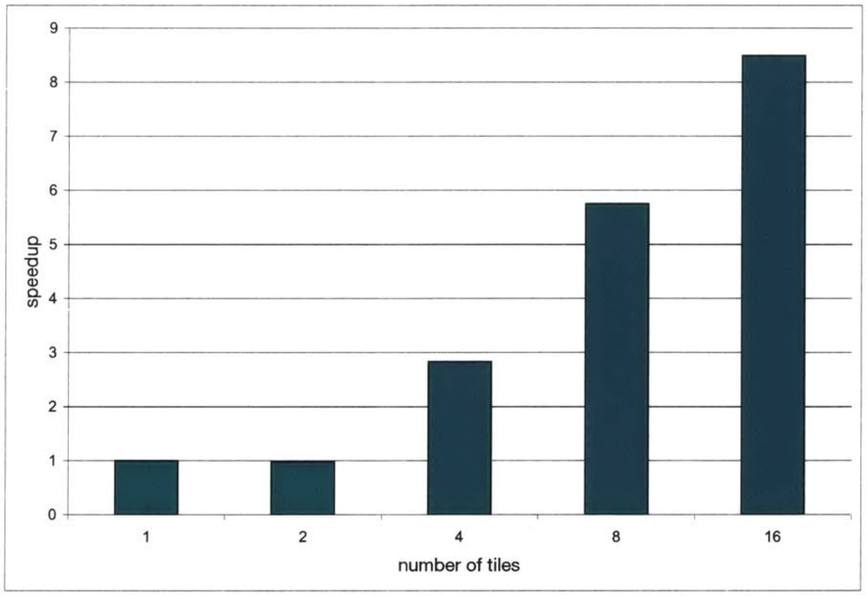


Figure 5-12: Speedup of row-oriented parallel matrix multiplication application for 1, 2, 4, 8, and 16 tiles.

Chapter 6

Conclusions and Future Work

This thesis presented rMPI, an MPI-compliant message passing library for the Raw tiled architecture. rMPI introduces robust, deadlock-free, and high-performance mechanisms to program Raw; offers an interface to Raw that is compatible with current MPI software; gives programmers already familiar with MPI an easy interface with which to program Raw; and gives programmers fine-grain control over their programs when trusting automatic parallelization tools is not desirable. Experimental evaluations have shown that the resulting library had relatively low overhead, scaled well with increasing message sizes for a number of collective algorithms, and enabled respectable speedups for real applications.

rMPI evolved considerably through three incarnations. Version I implemented, from scratch, a minimal set of the MPI standard, including the "core" six functions that allowed programs using basic MPI primitives to be run. Additionally, the design of Version I was somewhat inefficient. Version II expanded on Version I by implementing a larger set of the MPI standard in a more efficient manner. The final rMPI implementation, Version III, implements a large portion of the MPI standard in a robust, efficient, and extensible manner. While Versions I and II were written from scratch, Version III incorporated a portion of the LAM/MPI implementation, a complete, production-quality MPI implementation. The process of developing the three manifestations in this manner proved to be quite useful; the lessons learned and concepts grasped in the earlier implementations were ultimately necessary to construct the final implementation. Furthermore, the sheer size and complexity of the resulting system required such an incremental approach to develop an intuition for key ideas such as interrupt handler timing, efficient data structure architecture, and efficiently supporting extended functionality with low overhead.

There exist a number of potential improvements and enhancements to rMPI. First, rMPI currently does not support non-blocking sends and receives. Support for these mechanisms will be much more feasible when operating system support becomes available on Raw. Currently, rMPI supports word-aligned types, but not non-word-aligned offsets. Thus, rMPI could be extended to support arbitrary byte-aligned datatypes. If users of the current implementation of rMPI desire to send a non-word-aligned datatype, such as a `char`, they will have to place appropriate padding onto the message at the sender side, and remove the padding on the receiver side.

One optimization that was employed in Version I and Version II that was not carried forward in Version III is use of reusable packet buffers. As described in Section 3.2.2, the packet buffer pool scheme is used to minimize the amount of dynamic memory allocations; buffer space is dynamically allocated once, and reused by a number of potentially different users. This scheme mitigates the negative effects of many consecutive allocation and deallocation operations. This optimization was not employed in Version III because it would have added complexity to an already-quite-complex system. However, now that the system is stable, adding packet buffers could be a worthwhile enhancement.

It was clear in Section5.4.2 that the logarithmic broadcast algorithm performed noticeably better than the linear algorithm. This performance could most likely be improved even further if special collective communication algorithms were written specifically for Raw's dynamic network. In general, platform-specific collective communication algorithms constitutes a very rich research area that could provide substantial performance improvements.

In sum, rMPI achieved its goals of being MPI-Standard compliant, scalable, deadlock-free, high performance, and extensible. It held up under varying degrees of network congestion and provides a standard and straightforward interface with which to program Raw.

Bibliography

[1] Gdb: The gnu project debugger. http://www.gnu.org/software/gdb/gdb.html.

[2] An infiniband technology overview. Technology overview, InfiniBand Trade Association.

[3] Modelling superabsorbers and their interations with surrounding media. http://soft-pc3.zib.de/springer/neunzert_math.pdf.

[4] Moore's law 40th anniversary. http://www.intel.com/technology/mooreslaw/index.htm.

[5] Myrinet overview. Technology overview, Myricom, Inc.

[6] Open64 compiler tools. http://open64.sourceforge.net/.

[7] Top500 supercomputer sites. http://www.top500.org/.

[8] Valgrind debugging and profiling tools. http://valgrind.org/.

[9] Amarasinghe, S. P., Anderson, J. M., Wilson, C. S., Liao, S.-W., Murphy, B. R., French, R. S., Lam, M. S., and Hall, M. W. Multiprocessors from a software perspective. *IEEE Micro* (June 1996), 52–61.

[10] Burns, G., Daoud, R., and Vaigl, J. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium* (1994), pp. 379–386.

[11] Ciegis, R., and Zemitis, A. Numerical algorithms for simulation of the liquid transport in multilayered fleece. *15th IMACS World Congress on Scientific Computation and Applied Mathematics* (1997), 117–122. Wissenschaft und Technik Verlag Berlin.

[12] Forum, M. A message passing interface standard. Tech. rep., University of Tennessee, Knoxville, 1994.

[13] Forum, M. P. I. Mpi: A message-passing interface standard, 1995. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[14] Gropp, W. Tutorial on mpi: The message-passing interface. http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html.

[15] Gropp, W., and et al., S. H.-L. *MPI: The Complete Reference, Vol. 2.* The MIT Press, 1998.

[16] Hall, M. W., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam, M. S. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer* (December 1996).

[17] Jeffrey M. Squyres, Brian Barret, A. L. The system services interface (ssi) to lam/mpi. http://www.lam-mpi.org/papers/ssi/1.0-ssi-overview.pdf.

[18] Lee, W., Barua, R., and Frank, M. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct. 1998).

[19] Markoff, J. A new arms race to build the world's mightiest computer. *The New York Times* (Aug. 2005).

[20] Microsystems, S. Java 2 platform, standard edition (j2se). http://java.sun.com.

[21] Pacheco, P. S. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[22] Quinn, M. J. *Parallel Programming in C with MPI and OpenMP.* McGraw Hill, 2004.

[23] R., J., and Axelrod, T. *Programming Parallel Processors.* Addison-Wesley, 1988.

[24] Sankaran, S., Squyres, J. M., Barrett, B., and Lumsdaine, A. Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.

[25] Snir, M., and et al., S. O. *MPI: The Complete Reference, Vol. 1.* The MIT Press, 1998.

[26] Squyres, J. M., Barrett, B., and Lumsdaine, A. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[27] Squyres, J. M., Barrett, B., and Lumsdaine, A. The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department, 2003.

[28] Squyres, J. M., and Lumsdaine, A. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting* (Venice, Italy, September / October 2003), no. 2840 in Lecture Notes in Computer Science, Springer-Verlag, pp. 379–387.

[29] Taylor, M. The raw prototype design document, 2003. ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf.

[30] Taylor, M., Lee, W., Amarasinghe, S., and Agarwal, A. Scalar operand networks: On-chip interconnect for ilp in partitioned architectures. In *Proceedings of the INternational Symposium on High Performance Computer Architecture* (Feb. 2003).

[31] Taylor, M. B., Lee, W., Miller, J., Wentzlaff, D., Bratt, I., Greenwald, B., Hoffmann, H., Johnson, P., Kim, J., Psota, J., Saraf, A., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *Proceedings of International Symposium on Computer Architecture* (June 2004).

[32] Weickert, J. A mathematical model for diffusion and exchange phenomena in ultra napkins. *Math. Meth. Appl. Sci.*, 16 (1993), 759–777.

[33] William Gropp, Ewing Lusk, A. S. A high-performance, portable implementation of the mpi message passing interface standard. http://www-unix.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html.