Implementation of a Hardware-Optimized MPI Library for the SCMP Multiprocessor

Jeffrey Hyatt Poole

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

James M. Baker, Chair
Joseph G. Tront
Amitabh Mishra

July 21, 2004
Blacksburg, Virginia

Keywords: Parallel Architecture, Message-Passing Systems, Single-Chip Systems, Chip Multiprocessors, Message Passing Interface, MPI

Implementation of a Hardware-Optimized MPI Library for the SCMP Multiprocessor

Jeffrey Hyatt Poole

James M. Baker, Ph.D., Committee Chair

Department of Electrical and Computer Engineering

Abstract

As time progresses, computer architects continue to create faster and more complex microprocessors using techniques such as out-of-order execution, branch prediction, dynamic scheduling, and predication. While these techniques enable greater performance, they also increase the complexity and silicon area of the design. This creates larger development and testing times. The shrinking feature sizes associated with newer technology increase wire resistance and signal propagation delays, further complicating large designs. One potential solution is the Single-Chip Message-Passing (SCMP) Parallel Computer, developed at Virginia Tech. SCMP makes use of an architecture where a number of simple processors are tiled across a single chip and connected by a fast interconnection network. The system is designed to take advantage of thread-level parallelism and to keep wire traces short in preparation for even smaller integrated circuit feature sizes.

This thesis presents the implementation of the MPI (Message-Passing Interface) communications library on top of SCMP's hardware communication support. Emphasis is placed on the specific needs of this system with regards to MPI. For example, MPI is designed to operate between heterogeneous systems; however, in the SCMP environment such support is unnecessary and wastes resources. The SCMP network is also designed such that messages can be sent with very low latency, but with cooperative multitasking it is difficult to assure a timely response to messages. Finally, the low-level network primitives have no support for send operations that occur before the receiver is prepared and that functionality is necessary for MPI support.

Acknowledgments

I feel that a number of people who have helped me get to this point should be acknowledged for what they have done. First, I would like to thank my parents for supporting me in many ways throughout my academic career. I would like to thank the rest of the SCMP research group for not only their contributions to the project, but also their personal support as friends. My roommates also should be mentioned for asking me constantly when my thesis would be done, and thereby motivating me to work in order to silence them. I also need to thank my committee for putting time and effort towards my thesis and corresponding defense; there are certainly many demands on professors' time, and I appreciate that I was made a priority. Finally, I have to give extra thanks to my advisor whose guidance and insight into many of the problems that I faced has been truly invaluable.

This work has been supported by the National Science Foundation
through grant #CCR-0113948.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 SCMP

The SCMP (Single Chip Message-Passing) parallel computer system is a design that uses multiple processor cores on a single chip[1]. Each processor core has 2-8 MB of local memory, hardware support for multiple threads, and a network interface unit (NIU) for communicating with the rest of the processors. The basic components of each processor are shown in Figure 1.1. The processor cores are arranged in a grid and connected to neighbors in the four cardinal directions by an on-chip mesh network, as shown in Figure 1.2. This design serves a few different purposes.

First, it provides for easy thread-level parallelism. Instruction-level parallelism has its limits, and designers have reached a point of diminishing returns in trying to extract further instruction-level parallelism[2]. Thread-level parallelism is easier to make use of, and many applications are known to parallelize well. Even applications that were previously thought to have too high of a communication to computation ratio to parallelize well may be successful when the latency of the interconnect is sufficiently low, and SCMP's on-chip network creates

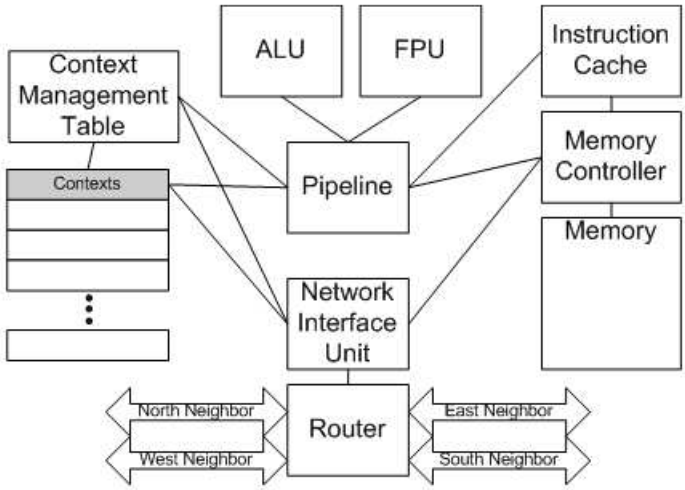

Figure 1.1: Internals of a SCMP Node

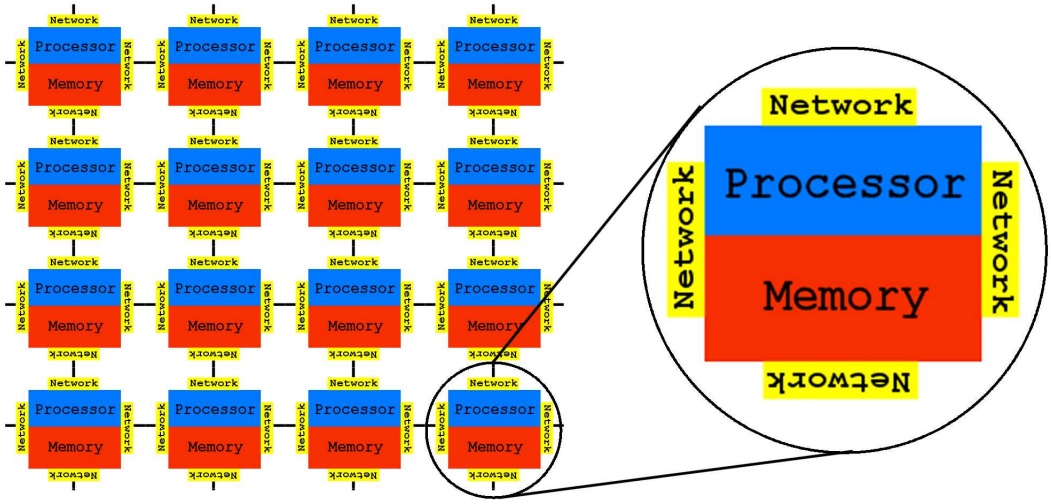

Figure 1.2: A SCMP 4x4 Grid of Processors

a potential for very low latency communications.

Second, processor design costs are rising[3][4]. Total design costs for processors increase as the architecture gets more complex and the number of transistors increase. These costs include development, verification, and testing costs. Silicon area is increasing faster than designers can reasonably make use of it, hence a trend towards larger caches, since they provide a performance improvement with very low design time. The ability to tile a design

is crucial to the minimal effort required for increasing cache size, and the same technique is used with SCMP. Instead of tiling cache elements, the SCMP design relies on tiling entire processors.

As semiconductor feature sizes decrease, transistors get faster and more of them can be placed on a chip of a given size. An unfortunate side effect of this decrease in feature size is that it requires a corresponding decrease in the width of wires connecting those transistors. Thinner wires have higher resistance and, as a result, higher propagation delays. The SCMP design compensates for this by minimizing wire lengths throughout the device. Individual processors are small, network connections are only provided to immediate neighbors, and each processor has its own memory, keeping memory access times short.

By creating a simple RISC processor that may not perform as well as a modern desktop processor and then laying many of them down on a single chip, this design takes advantage of the short design time of a simple processor. Yet, it has the potential to surpass modern desktop processors in performance. In this case, the small size of the design is advantageous, as it allows more processors to be tiled on a single chip. Instead of using a large amount of silicon space and designer time to add features that provide minimal performance improvements, that space can be filled with more processor cores, and current performance numbers show this to be a promising ideology.

1.2 MPI

The MPI (Message Passing Interface) communications library[5] is designed for use with high-performance computing on clusters and multiprocessor systems. In fact, it is completely unsuitable as a general communications library as most implementations require knowledge of the entire network before beginning and different implementations are not interoperable (however, there is an "Interoperable MPI" standard as of 2000[6]). As a parallel commu-

nications library, MPI provides an impressive library of functions for moving data between cooperating processes. The intention of MPI is to provide a platform for the creation of high-performance parallel applications that can allow them to be ported to many different parallel systems.

This is considered important for two reasons. First, it allows companies to create parallel applications that can be easily ported between different systems allowing them to benchmark applications on multiple systems. The ability to do this allows reasonably fair comparisons between different hardware configurations. Second, it allows users to change their hardware and operating environment as time goes on without rewriting their software. This flexibility means that users are not locked into a hardware environment once the software is written. It also allows for the creation of parallel libraries and benchmarks that can be run unmodified on a number of different systems thus preventing work from being duplicated for every different system.

At this point, MPI is arguably the most popular parallel communications library, especially for distributed memory systems. Its popularity is due to its portability and its flexibility in supporting a number of communications paradigms. It also has a large emphasis on performance and whenever possible it is designed to allow optimizations such as in-place transfers from memory to the network. With a standard consisting of over 120 functions[5], it is designed to be a powerful and flexible library. This thesis will explain many of the details of the MPI standard in Chapter 3.

1.3 Rationale for MPI on SCMP

The design of a new processing architecture requires a tremendous amount of analysis after the initial design is complete. Not every design that looks good on paper will actually show a performance improvement. All designs require analysis to determine applications for

which the platform performs well or poorly. There are many ways to make such analysis, including purely theoretical analysis, synthetic benchmarks, and actual applications. While theoretical analysis and synthetic benchmarks are useful in the early stages of the design, actual applications must be run on the design to get a complete picture of the performance capabilities and bottlenecks.

A number of applications have been ported to the SCMP architecture in an attempt to determine its performance, but porting can be a long and tedious process. One of the greatest barriers to efficient porting is that the on-chip network provides only the most basic communications facilities. The native SCMP network support is limited to writing data to remote memory locations and invoking remote methods as a new thread on the remote processor. This means that parallel programs using more advanced libraries require significant effort to convert.

One such communications library for parallel systems is MPI, as mentioned above in Section 1.2. The research team decided that creating an implementation of MPI for the SCMP platform would allow quick porting of existing MPI applications as well as providing rich communications functionality for future development. Such functionality would ease porting existing applications using other libraries and also would make original programs easier to develop. Providing a message-based communications library in addition to the fast message support already present would increase the options available to the SCMP developer.

Message-passing systems are often considered to be more difficult to write software for than shared-memory systems, which mimic the memory semantics of the single processor systems that most programmers learned to write software on. The need to explicitly move data around requires more effort on the part of the programmer. However, the requirement of explicitly planning the data movement can lead to more efficient software since it is obvious which instructions will require communications activity. By creating a more sophisticated message-passing library to be used on the SCMP system, it is hoped that developers will be

more comfortable with the message-passing paradigm.

Using a high-level communications library such as MPI instead of the native SCMP messaging support will undoubtedly come with a performance hit. Any form of abstraction will always have overhead. The ability to send a message without knowing if the destination is prepared to receive it implies that either the sender and receiver must communicate and determine if the message can be sent at the current time, or the receiver needs to allocate temporary storage to put the message in and copy it into the real destination once that is known. While overhead is unavoidable, an implementation can be designed to take advantage of features of the underlying hardware to minimize that overhead.

This thesis discusses the creation of an MPI implementation for the SCMP hardware design. This implementation will take into account the architecture of the SCMP design in an attempt to provide sufficiently high performance that MPI can be used in place of the standard SCMP network primitives in some cases without changing the general performance relationship between this system and other machines. Since SCMP hardware has not yet been fabricated, a cycle-accurate hardware simulator will be used for performance benchmarking purposes.

Chapter 2

Background and Related Work

This section will quickly describe some background information for this work. First, it will provide a brief survey of parallel communications libraries. Next, it will describe the SCMP network support since an understanding of the communications ability of the hardware is important to understanding the design decisions made during the design of this communications library. Finally, some related work into additional hardware communications support will be described.

2.1 Parallel Communications Libraries

Over time, a number of communications libraries have been developed for use in parallel processing applications. The following are three language-neutral communications libraries that are well-known in the high-performance computing community. There are also a number of language-specific libraries including Co-Array Fortran, High Performance Fortran, and Unified Parallel C; however, they have not been described here for the sake of brevity.

2.1.1 Active Messages

Active Messages are often referred to as a "RISC approach to communication," due to an API that provides simple primitives for communication which, by design, map efficiently onto hardware and thereby meet the requirements of higher-level communications libraries. While Active Messages could be used as a parallel communications library by itself, it is more often used to implement a higher-level communications library such as PVM or MPI. As described in [7]:

> Active Messages is an asynchronous communication mechanism intended to expose the full hardware flexibility and performance of modern interconnection networks. The underlying idea is simple: each message contains at its head the address of a user-level handler which is executed on message arrival with the message body as argument. The role of the handler is to get the message out of the network and into the computation ongoing on the processing node. The handler must execute quickly and to completion. ... this corresponds closely to the hardware capabilities in most message passing multiprocessors where a privileged interrupt handler is executed on message arrival, and represents a useful restriction on message driven processors.

The inherent SCMP network support was patterned somewhat in the style of Active Messages. When later sections describe the SCMP network support, notice the similarities between Active Messages and SCMP THREAD messages — both provide the address of a handler function and each word of data in the message is passed as a parameter to that function. Active Messages also provides a simple interface on top of which more advanced libraries can be built.

2.1.2 Parallel Virtual Machine (PVM)

PVM (Parallel Virtual Machine) is a message-passing communications library designed to link multiple processing resources into one "virtual machine." Implemented by running PVM daemons on each physical machine, the virtual machine effectively provides a distributed operating system. PVM was created by a single research group at Oak Ridge National Laboratory (ORNL) in 1989 as they developed an implementation of the library[8]. Many of its features evolved from incremental improvements requested by users. PVM came to be the prevailing standard before the creation of MPI, which is discussed next.

The design of PVM aimed for portability over performance. It was developed with a focus on networked clusters of heterogeneous machines, which implied a communications fabric of low performance LANs or possibly the Internet. Also, the goals of the project included scaling, fault tolerance, and heterogeneity of the virtual machine rather than process speed. PVM 3.0 was released in 1993 with an updated API to provide support for virtual machines of multiple large multiprocessors.

2.1.3 Message Passing Interface (MPI)

MPI (Message Passing Interface) is a message-passing communications library designed by the MPI Forum, a group made up of a diverse selection of implementers, library writers, and end users. This development structure imbued the library with a well-defined, object-oriented design from the very first implementation. MPI's use of opaque objects (objects whose contents are not visible to the user) allows extensions to the standard without breaking existing code.

The MPI Forum was first started in April 1992, a first draft of the MPI standard was presented in 1993, and it was released in 1995. Started by massively parallel processor

(MPP) vendors to avoid each creating incompatible standards, the standard emphasizes high performance within a portable design. Features include a large set of point-to-point and collective communications routines, communications contexts, and derived datatypes that support messages of noncontiguous data. MPI will be covered in detail in Chapter 3.

2.2 SCMP Network

The primary advantage from putting multiple processors on a single piece of silicon is the improved communications abilities between processors. Some applications have so many dependencies within themselves that turning the program into multiple threads of execution requires too much communication to be efficient. Either the communication required may serialize the different threads of execution, or the threads spend so much time waiting for data to get from one thread to another that the computation time is overshadowed by the communications time.

If the communication serializes the threads, not much can be done aside from trying to decompose the program in a different manner. However, if the communications time is too long, it is possible that a higher bandwidth or lower latency communications network could make the existing program partitioning acceptable. The SCMP multiprocessor is designed to take advantage of its low-latency, high-bandwidth on-chip communications to enable parallelization of processes that may have been inefficient to parallelize before.

2.2.1 Network Hardware

The SCMP network is a grid network where every processing element is connected to a router which is, in turn, connected to the north, south, east, and west neighbors. One of the goals of the SCMP design is to keep wire lengths small to minimize latency and skew

Head Flit	Address Flit	Data Flit	...	Data Flit	Tail Flit

Figure 2.1: Flits in an SCMP Network Message

Valid	Head=1	Tail	X Offset x 4	Y Offset x 4	THREAD/DATA	Stride x 23

Figure 2.2: Head Flit Structure

considerations, which makes non-neighbor network connections undesirable. An emphasis on keeping wire lengths short is partially due to the shrinking of wire diameters accompanying the traditional shrinking of transistors as fabrication processes improve. Thinner wires result in higher resistance, which translates into greater propagation delay[9]. The router uses a pipelined design to improve throughput and wormhole switching to reduce latency.[10]

Wormhole switching involves breaking a message up into "flits" or flow-control digits. Messages in general are made up of a head flit, data flits, and a tail flit. The SCMP design uses the first data flit as a special "address" flit. The layout of a message is shown in Figure 2.1.

The key to wormhole switching is that the head flit contains all the information needed to route the message. This way, once the routing is determined for the head flit, it can be sent along that path and the router knows to send every other flit that came over the link used by the head flit the same way until a tail flit is seen. At that point, it prepares for a head flit from another message. This is opposed to "store and forward" routing which waits for an entire message then passes that message on. Wormhole switching has much lower latency because it pipelines transmission of the parts of a message.

Valid	Head	Tail	Data x 32

Figure 2.3: Data Flit Structure

Flits in the SCMP network are actually 35 bits wide, but only carry a data payload of 32 bits. The use of the other bits is shown above in Figure 2.2 and Figure 2.3. Both structures have a Head bit, which is 1 when the flit is a head flit, and a Tail bit, which is 1 when the flit is a tail flit. Also a Valid bit is used by routers to determine if a flit is coming across a link from another router. The Valid bit is added when the flits enter the network.

The head flit contains the address of the destination processor as an X offset and a Y offset. Messages are routed in the SCMP network using dimension-order routing. Dimension order routing is a deadlock-free routing algorithm that causes data to be routed along one dimension until it is at the correct location in that dimension and then to be routed along the next dimension until it reaches its destination. Using offsets further simplifies the process.

The first bit of the offset is the direction along that dimension and the rest of the bits are devoted to the number of hops in that direction. The SCMP router routes in the X dimension first, so if the last three bits of the X offset are not '000', then it knows it needs to route in the X dimension. The first bit tells it which direction it needs to go and, therefore, which output channel it will use. Finally, before the message is passed on, the router decrements the last three bits of the offset. If a router receives a message with zero magnitude X and Y offsets, it knows to route it to the injection/ejection channel that goes to the NIU (Network Interface Unit) of the node associated with the router. This system is especially elegant because it allows the routers to operate without context — a router does not need to know where it is located or how to get to a specific destination node.

SCMP supports two different kinds of network messages: THREAD and DATA. THREAD messages cause allocation of a thread at the destination, setting the instruction pointer (IP) to the address sent in the second flit of the message, and loading the registers with values from the flits after that. That thread will not execute immediately, but instead will have to wait until threads before it in the queue give up the processor. THREAD messages ignore the stride value. DATA messages cause the destination processor to use a DMA transfer to

write the data from the body of the message into the processor's memory. The "stride" value causes the receiving processor to place incoming words "stride" words apart in memory. This is especially useful for sending columns from a 2-dimensional array in a language that stores arrays in row-major order (like C).

A final feature of the SCMP network hardware is the use of virtual channels. Virtual channels allow different messages to be multiplexed over the physical channels. Typically, only one message can use a physical link at a time; however, virtual channels provide the appearance of multiple physical links over a single physical link in order to prevent congestion. This is important in wormhole switching because whenever a message gets blocked for some reason, it continues to occupy much or all of the path from its source to its destination. These occupied links cannot be used by other messages while that message is blocked in the network. Virtual channels allow messages that are able to make progress to do so while other messages are blocked over the same links.

2.2.2 Network Instructions

The SCMP hardware has some built-in machine level network instructions in order to facilitate efficient communication between processors. A list of instructions is provided in Table 2.1. The `SENDH_I` and `SENDH_R` instructions create the head flit and have the NIU (Network Interface Unit) inject it into the network. The rest of the instructions send data, and possibly set the tail bit on the last flit to signal the end of the message. Note that the instructions are at the abstraction level of creating flits, instead of a higher level, such as sending whole messages. This allows messages to be built on the fly straight into the network; however, it also increases the time spent to transfer a message if computation occurs between sending flits, which can increase network contention.

One feature worth noting is that when a DATA message is sent with a stride value, this does

not necessarily correspond to the stride as used on the source processor. This allows for some interesting capabilities, including the ability to transpose matrices by sending a row (stride=1) and having it stored as a column (stride = number of columns) or vice-versa. This can also be useful if data for a problem is decomposed by giving one item to each processor. In this case, a processor could distribute data by using a local stride equal to the number of processors and a destination stride of 1. Then this data could be sent back using a local stride of 1 and a destination stride equal to the number of processors.

At the level of C code, there is yet another level of abstraction. The C commands, however, are designed to be very efficient. They only add more functionality in order to compensate for the function call overhead (if each assembly instruction was wrapped in a C function, the function call overhead would dominate the communication time). A list of thread-related functions is in Table 2.2, and a list of data-related functions is in Table 2.3.

Table 2.1: SCMP's Network Instructions

Instruction	Parameters	Explanation
`SENDH_I`	dest(r), type(i), addr(i)	Sends a head flit, which begins the transmission of a message. The type parameter determines if the message is a THREAD or DATA message. The immediate address parameter is limited to 17 bits.
`SENDH_R`	dest(r), type(i), addr(r), stride(i)	Sends a head flit, much like `SENDH_I`. This version uses a register for the address parameter to allow larger addresses.
`SEND2`	data1(r), data2(r)	Sends two data flits, one from each register.
`SENDE`	data(r)	Sends a data flit with the tail bit set to end the message.
`SEND2E`	data1(r), data2(r)	Sends two data flits with the tail bit set on the second one to end the message.
`SENDM`	addr(r), stride/count(r)	Tells the NIU to send data directly from memory using a DMA transfer. The upper 16 bits of the second parameter provide the stride value and the lower 16 bits provide the number of data items to send.
`SENDME`	addr(r), stride/count(r)	Works like `SENDM`, except the last flit will have the tail bit set to end the message.

(r) - Register Parameter / (i) - Immediate Parameter

Table 2.2: SCMP's C Network Functions — THREAD

Function	Parameters	Explanation
createThread	`int dst_node`, `void(*addr)()`, `void(*callback)()`, `...`	Creates a thread on the destination node `dst_node` calling the function at `addr`, passing all additional parameters to that function, then calling the function `callback` on the local node when done.
parExecute	`int num_nodes`, `void(*addr)()`, `...`	Creates a thread on `num_nodes` nodes calling the function at `addr` and passing the additional parameters to the function. This also waits for all threads to complete before continuing.
getBlock	`unsigned node_id`, `char *dst_addr`, `unsigned dst_stride`, `char **src_addr`, `unsigned src_offset`, `unsigned src_stride`, `unsigned num_words`	Creates a thread on node `node_id` that will send the data message specified in the parameters. This is needed since the network library only supports sends explicitly, not receives.

Table 2.3: SCMP's C Network Functions — DATA

Function	Parameters	Explanation
sendDataIntValue	`int dst_node`, `int *dst_addr`, `int value`	Sends a single integer to be written to `dst_addr` on node `dst_node`.
sendDataFloatValue	`int dst_node`, `double *dst_addr`, `double value`	Sends a double floating point value to be written to `dst_addr` on node `dst_node`.
sendDataBlock	`int dst_node`, `int *dst_addr`, `int dst_stride`, `int *src_addr`, `int src_stride`, `int count`	Sends a block of values `count` words long from address `src_addr` on the local node to address `dst_addr` on node `dst_node`, using `src_stride` and `dst_stride` to determine the spacing of the source and destination values. It blocks until the transfer completes.
sendDataBlockNB	`int dst_node`, `int *dst_addr`, `int dst_stride`, `int *src_addr`, `int src_stride`, `int count`	This is a non-blocking version of *sendDataBlock()*.

The most common network functions used in C are *createThread()*, *sendDataIntValue()*, and *sendDataBlock()*. *createThread()* is used to send a THREAD message to execute code on a remote processor. Starting a thread on another processor is the easiest way to send control messages between processors. DATA messages can overwrite each other and cause a variety of other problems when used as control messages. *sendDataIntValue()* is a simple function that sends a single integer to a remote process using a DATA message. This is often used for synchronization to set a flag. Finally, *sendDataBlock()* is used to send a block of data from one processor to another using a DATA message. This function supports destination and source strides, and it uses the `SENDM` primitive for most of the communication, freeing the processor to do other things while the data is being sent via DMA.

2.2.3 Characteristics

Three important characteristics of the SCMP network from a design standpoint are as follows:

1. THREAD messages are virtually the only kind of message that can be used in control situations
2. Data can be transferred in the background using DMA
3. The on-chip network is designed to be able to send 2 flits of data per processor clock cycle.

Note that the ability to inject two flits (64 bits) of data into the network per clock cycle implies that time spent on computation will dramatically increase the data transfer time, since data can be transferred so quickly. These characteristics will be explained in further detail in Section 4.1.

2.3 SCMP Hardware Messages

Charles W. Lewis, Jr. has made progress in implementing hardware-based messages on the SCMP hardware[11]. This functionality was being developed in parallel with the MPI implementation that is at the heart of this thesis. However, one of the prime motivations for creating hardware message support is to enable the implementation of communications libraries such as MPI more efficiently. Since this project was not complete during that work, analysis of the MPICH[12] library was used to help determine the requirements of such hardware support. As a result, this work should be discussed here for completeness.

The MPI library provides an API for full send/receive message semantics, and it makes sense to explore hardware support for this communication paradigm. When creating a software solution for message semantics, it becomes clear that hardware support could overcome a few problems encountered in the purely software approach. While these difficulties will be explained further in later sections, suffice to say that sending a message to a process that is not prepared to receive it requires some overhead that would be unnecessary if there were hardware support for that operation.

The proposed hardware message support includes options for both ready-mode sends (where the receiver is required to be ready for the receive before the send operation occurs) and rendezvous-mode sends (where a handshake occurs to prevent the sender from sending before the receiver is ready). Since ready-mode sends are not supported in this MPI implementation, only rendezvous-mode sends will be discussed. This hardware scheme effectively allows the NIUs (Network Interface Units) of the sender and receiver to perform the handshake operation instead of requiring software support. The sender sends a RTS (Request To Send) message to the receiver, when the receiver is ready it sends a CTS (Clear To Send) message back to the sender, and then the sender starts transferring the message to the receiver.

With this sort of support, the network round-trip-time becomes the limiting factor for the

latency associated with sending messages. Performing this service in hardware instead of software, at a minimum, removes the processor cycles dedicated to creating a thread to perform the operation and the code to perform the operation itself. It can also drastically decrease latencies due to processors currently performing an operation and not allowing these newly-created threads a chance to execute.

Chapter 3

Introduction to MPI

The MPI communications library is a complex piece of software. The MPI 1.2 standard, which the implementation in this thesis uses as a guide, consists of approximately 120 functions. This chapter is not an exhaustive reference — an implementor or user of MPI would need something more detailed — but it is designed to provide the reader with knowledge of the basic structure of MPI and the most commonly used functions in the standard. A casual reader may find it more useful to try to skim this section instead of attempting to absorb all of the details.

This chapter starts out explaining the MPI concepts of Groups and Communicators, along with some common functions for manipulating these objects. After that, MPI's support for user-defined datatypes will be explained. Finally, point-to-point communication and collective communication functions will be described.

3.1 Groups

Most communications libraries have the concept of a group of processes. In MPI, a group of processes is represented by an opaque `MPI_Group` data structure. Most of the functionality associated with groups is fairly straightforward, so this shall be a brief overview.

First, the functions *MPI_Group_size()* and *MPI_Group_rank()* are used by a process to determine the size of a group and what its rank within the group is (if it is not a member of the group, the function returns `MPI_UNDEFINED`). The function *MPI_Group_translate_ranks()* can be used to determine the ranks of processes in one group if their ranks are known in another group. For example, if the ranks of some processes are known in the group associated with `MPI_COMM_WORLD` (the default communicator, which will be discussed in the next section), this information can be used to determine the ranks of those processes in another group. This function will return `MPI_UNDEFINED` in the results array for any process listed that is not in the second group, so it can also be used to determine what processes are in that group.

MPI_Group_compare() will compare two different groups and determine if they are identical (`MPI_IDENT`), similar (`MPI_SIMILAR`), or unequal (`MPI_UNEQUAL`). In this context, two groups are identical if they have the same members in the same order and similar if they have the same members but in a different order. The function *MPI_Comm_group()* is used to determine the group associated with a communicator (again, communicators will be explained in the next section).

Finally, there are a number of set-like operations available on groups. The functions that perform these operations are *MPI_Group_union()* (which returns the set union of two groups, with the elements of the first group first), *MPI_Group_intersection()* (which returns the set intersection of two groups, ordered as in the first group), and *MPI_Group_difference()* (which returns all elements in the first group but not in the second). There are some more group modification functions, such as *MPI_Group_incl()* (which lets you specify which members of

one group you want in a new group), *MPI_Group_excl()* (which allows duplication of a group with certain members excluded), and *MPI_Group_range_incl()* and *MPI_Group_range_excl()* (which perform much like the last two functions, except these allow you to specify ranges of processes). Finally, there is *MPI_Group_free()* to deallocate a group.

3.2 Communicators

MPI introduces the notion of communicators. Communicators are opaque objects (of type `MPI_Comm`) that represent a specific communications domain. Communicators are associated with a group of processes, but there may be multiple communicators that are associated with the same or overlapping groups. All communicators have a size, which is the number of processes in the group associated with it. Every process has a unique numeric rank in every communicator of which it is a part. This is the same as its rank in the group associated with the communicator. Every communications operation must be associated with a communicator.

The driving force behind the use of communicators instead of other methods of keeping unrelated communications separate is that a library can take a provided communicator, duplicate it, and ensure that it can communicate with all of the processes belonging to the original communicator without those messages being confused with messages from the program itself. MPI supports the notion of supplying each message with a numeric "tag" so receives are only matched with sends that use the same tag. However, there is no way for libraries to know which tags are in use by the program, and therefore there is no way for a library creator to safely separate their communication from that of the base program using tags alone. Even setting aside a tag space for libraries, if one library called another library, the same problem remains.

When an MPI program starts, one communicator is created by default, called `MPI_COMM_WORLD`.

This communicator represents the set of all of the processes that are part of the system. Additional communications domains can be created by calling *MPI_Comm_dup()* to create a new communicator with the same group of processes as before, by calling *MPI_Comm_create()* with a group of processes that should be included in a new communicator, or by calling *MPI_Comm_split()* to break an existing communicator into a set of new communicators with disjoint subsets of processors.

MPI_Comm_split() is especially flexible and requires further discussion here. It takes a base communicator, a "color", a "key", and it returns into a pointer to either a new communicator that contains this process or `MPI_COMM_NULL`. The "color" and "key" parameters may be, and usually are, different on every process. Every process with the same "color" parameter ends up in the same new communicator produced by this function. The "key" parameter determines the ordering of the ranks of the processes in the new communicator — ties are broken by using the processes' rank in the old communicator. The following example demonstrates how this command could be used to produce a communicator for every process in `MPI_COMM_WORLD` that contains only members of its column. If called on a 4x4 grid of processors, it would create 4 communicators consisting of the following processors: (0,4,8,12), (1,5,9,13), (2,6,10,14), and (3,7,11,15).

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_split(MPI_COMM_WORLD, myRank % getXDim(), myRank, &column_comm);
```

It is also worth nothing that the functionality of *MPI_Comm_dup()* can be emulated with the following command. Note how the "color" and "key" parameters are the same on every process.

```
MPI_Comm_split(MPI_COMM_WORLD, 1, 1, &duplicate_comm_world);
```

The remaining communicator functions are all fairly straightforward. *MPI_Comm_size()*

and *MPI_Comm_rank()* are simply shortcuts for *MPI_Group_size()* and *MPI_Group_rank()* called on the group associated with the communicator. The function *MPI_Comm_free()* can be used to free the memory associated with a communicator that is no longer in use. *MPI_Comm_compare()* can be used to compare two communicators in a similar manner to how *MPI_Group_compare()* compares groups, except that this function can return `MPI_IDENT` (if they represent the same communications domain), `MPI_CONGRUENT` (if the communications domains are different, but the groups are identical), `MPI_SIMILAR` (if the associated groups have the same members, but not in the same order), or `MPI_UNEQUAL` (if the groups are different).

3.3 Type functions

The MPI standard provides a rich set of features for manipulating datatypes. MPI comes with a full set of predefined datatypes, some of which are listed with their C equivalents in Table 3.1. Note that all datatypes are of type `MPI_Datatype`. Two types in particular need some explanation. First, `MPI_BYTE` is not defined as a C type because different machines may interpret characters differently, but a byte is specifically 8 bits of data without interpretation, so `char` is not a valid type to define `MPI_BYTE`. The other type that needs explanation is `MPI_PACKED`. This datatype is the result of using the *MPI_Pack()* function, which will be explained shortly.

Despite providing a good variety of predefined datatypes, MPI also provides a number of functions for creating new datatypes. Before getting into those functions, however, some concepts MPI has regarding types need to be explained. Every datatype in MPI has a lower bound, an upper bound, and an extent. These properties are related by $extent = upper\_bound - lower\_bound$. Datatypes also have a size, which is a distinct concept from the extent.

Table 3.1: Some Predefined MPI Datatypes

MPI Datatype	C Equivalent
`MPI_CHAR`	`signed char`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_SHORT`	`signed short`
`MPI_UNSIGNED_SHORT`	`unsigned short`
`MPI_BYTE`	n/a
`MPI_INT`	`signed int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_LONG`	`signed long`
`MPI_UNSIGNED_LONG`	`unsigned long`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_PACKED`	n/a

In most datatypes, the lower bound is 0 and therefore the upper bound is equal to the extent. However, MPI allows the lower bound to be explicitly set to positive or negative numbers. For example, a user could create a datatype with a lower bound of -5. If the user then tried to send an element of this type from a buffer located at address 100, MPI would start reading the data from address 95. If the user had set the lower bound to 5, MPI would have started reading the data from address 105.

The concept of the extent is important as well. For example, assume a machine that requires doubles to be aligned at addresses that are multiples of 8. Suppose a type was defined with a `double` at offset 0 and a `char` at offset 8. The notation `{(double,0),(char,8)}` will be used to represent this datatype. The total size of the datatype would be 9, since MPI uses the term size to mean the total size of the data in the message. However, placing two of

these structures in memory would require more than 18 bytes due to alignment restrictions. As a result, the extent of the datatype is 16, since 7 bytes of spacing would be required. To tie this in with the discussion of lower and upper bounds, the lower bound of this datatype would be 0 (since it starts at offset 0) and the upper bound would be 16 (since that is the sum of the lower bound and extent).

With that background information, the discussion of datatype creation can begin. There is a function, *MPI_Type_dup()*, that will duplicate a datatype, but this is of limited use to most people. The *MPI_Type_contiguous()* function can create a new datatype that represents an array of elements of another datatype. It takes the number of elements wanted in the array, the base datatype to create the array from, and a pointer to the new datatype. For example, `MPI_Type_contiguous(5, MPI_INT, &int_array);` would create a type called `int_array` which could be used to transmit an array of 5 integers. Note that sending 5 `MPI_INT` objects from a buffer is identical to sending 1 `int_array` object, and that it is legal to use them interchangeably (i.e., you can send 5 `MPI_INT` objects from one process and receive it as 1 `int_array` object at its destination). Note that *MPI_Type_contiguous()* uses the extent of the base datatype to determine the spacing between elements.

MPI_Type_vector() is similar to *MPI_Type_contiguous()*, except that it also takes "blocklength" and "stride" parameters. This function creates an array of "count" blocks, each of which is "blocklength" elements long, and which have "stride" elements between the start of adjacent blocks. For example, calling this function with a count of 2, a blocklength of 2, a stride of 3, and a base type of `MPI_INT` (which we will assume has an extent of 4 bytes), would produce the datatype `{(int,0),(int,4),(int,12),(int,16)}`. Note that there are two integers adjacent to each other, then a space the size of an integer, then two more integers adjacent to each other. The two blocks of two integers started three integers apart. The *MPI_Type_create_hvector()* function is the same except the stride is specified in bytes instead of multiples of the base datatype.

The *MPI_Type_indexed()* function allows a user to specify a data layout much like *MPI_Type_vector()*, except that it takes an array for the blocklength parameter and, instead of specifying a stride, an array of displacements is used. This allows layouts where the blocks are of different lengths and the spacing between the blocks is not consistent. The "count" parameter specifies how many elements are in the arrays. *MPI_Type_create_indexed_block()* has similar functionality, except that it allows the user to specify a single blocklength that will be used for all blocks. *MPI_Type_create_hindexed()* changes *MPI_Type_indexed()* to use byte displacements instead of using a number of elements.

MPI_Type_create_struct() is effectively an extension of *MPI_Type_create_hindexed()* that takes an array of datatypes instead of a single base datatype. The displacements are specified in bytes, since number of elements becomes ambiguous when dealing with multiple datatypes. This function has the power to define datatypes even greater in complexity than the C `struct` keyword is able to define since it allows the user to explicitly list the displacement of each element. Every type creation function described thus far can be implemented with *MPI_Type_create_struct()*.

After a custom datatype is created, the user must call *MPI_Type_commit()* on the type before it can be used in a communication. *MPI_Type_free()* can be called on a datatype once it is no longer needed.

One of the main uses of these custom datatypes is to be able to send non-contiguous data as easily as sending predefined datatypes. For example, a user could easily define a datatype that would allow them to send every 3rd integer from an array. In communications libraries that do not allow such flexible creation of datatypes, the standard technique is to "pack" the data into a contiguous buffer before sending it. To allow users to continue to use this paradigm, MPI provides *MPI_Pack()* and the associated *MPI_Unpack()*. These functions are also more efficient in some cases than explicit datatype creation, especially when the structure of the data is not known beforehand and that structure would need to be communicated to

the recipient along with the data.

MPI_Pack() allows the user to "pack" elements into a buffer, then send it as an element of type `MPI_PACKED`. When received on the other end, the user can "unpack" the constituent elements by using the *MPI_Unpack()* function. MPI also provides *MPI_Pack_size()*, which will calculate the amount of space required to pack something and, thus, allow the user to determine an appropriate buffer size.

3.4 Send/Receive Semantics

The basic MPI primitives for sending and receiving messages are *MPI_Send()* and *MPI_Recv()*. Both take the following parameters: a buffer to read data from or write data to, the number of elements to send/receive, the datatype of the elements to send/receive, the source/destination address, a numeric tag used to prevent similar sends/receives from being confused with each other, and the communicator that this transfer should use. The *MPI_Recv()* command also allows a user to provide a pointer to an *MPI_Status* object so that it can return some information about the transfer.

One of the defining characteristics that shapes the implementation of these routines is that while *MPI_Send()* requires a destination and a tag, the *MPI_Recv()* command can be passed the constants *MPI_ANY_SOURCE* and/or *MPI_ANY_TAG* as the source and tag parameters, respectively. This means that the receiver may not even know where the message is coming from. As a result, in the general case, messages are "pushed" by the sender to the receiver instead of having the receiver request the message from the sender.

Another characteristic to note is that there are non-blocking versions of both functions (*MPI_Isend()* and *MPI_Irecv()*). Messages sent by nonblocking sends may be received by blocking receives and vice-versa. This means that there must be support for receiving mes-

sages out-of-order since the message you want to receive may not be the first message you will get.

There are three versions of each send command that explicitly specify the mode of operation for the transmission. The normal *MPI_Send()* and *MPI_Isend()* calls use the "standard" mode, which is the implementation default. The standard mode is considered to be a "non-local" operation because it is not guaranteed to be able to complete without communicating to other processes. The normal receive calls will receive data sent by any of the send modes.

The first explicit mode provided is the "buffered" mode, which is specified by using the *MPI_Bsend()* and *MPI_Ibsend()* calls. A buffered send operation can be started regardless of whether a matching receive has been posted and may even complete before a matching receive is posted. A buffered send is considered a "local" operation because it's completion does not rely on any other processes, so it can be completed without communication. MPI requires that the application provide explicitly allocated memory to buffer the messages.

The next explicit mode is the "synchronous" mode, which is specified by using the *MPI_Ssend()* and *MPI_Issend()* calls. This mode can be started if a matching receive has not been posted; however, it will not complete successfully until a matching receive is posted and the receive operation has started to receive the message. As a result, the completion of the send call indicates not only that the receiver has matched this send with a waiting receive, but that the send buffer can be reused. A synchronous mode send is non-local.

The final explicit mode is the "ready" mode, which is specified by using the *MPI_Rsend()* and *MPI_Irsend()* calls. A ready-mode send may only be started if the matching receive has already been posted. If this is not the case, the call is considered to be erroneous and the outcome is undefined. A ready-mode send has the same semantics as the standard-mode send, except that in some implementations the use of ready-mode may eliminate the need for a handshake, so it may provide a performance advantage over the standard mode.

3.5 Collective Routines

The term "collective" refers to communications operations that communicate between multiple processes simultaneously. MPI provides a vast array of collective communication routines, including ones for broadcasting, spreading, and collecting data. MPI can also perform reduction and scan operations using either pre-defined or user-defined operations. Collective routines must be called by all members of the group associated with the communicator used; however, the effect may be different on different processes. MPI collective routines are not required to synchronize the processes although they may do so. This means that the MPI programmer must ensure that there are no cyclic dependencies between calls (as this may lead to deadlock in a synchronizing implementation) and also ensure that if synchronization is needed, it is explicitly provided via a barrier or similar routine.

3.5.1 *MPI_Barrier()*

MPI's barrier routine takes only one parameter – a communicator object. This routine simply blocks when entered until all members of that communicator have arrived, at which point they are all allowed to proceed. Barriers are a common function in parallel programming because they provide a simple way to synchronize multiple processes.

3.5.2 *MPI_Bcast()*

MPI_Bcast() is a function that broadcasts data from one process to many other processes (shown in Figure 3.1). It is called by the process sending the data as well as every process receiving the data. The source of the data is determined by a parameter that must be the same on all systems defining the "root" node. If a node calls *MPI_Bcast()* and specifies its own rank in the communicator as the root parameter, it sends data from its buffer parameter

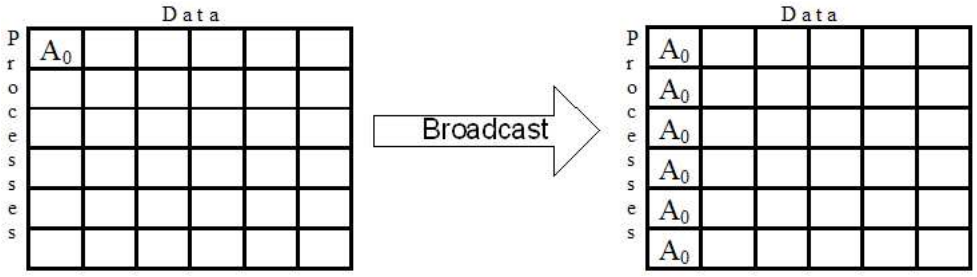

Figure 3.1: Broadcast Example

to every other node in the communicator. If a node's rank in the communicator is not equal to the root parameter, it receives data into its buffer parameter from the root node. The result is that the root's buffer is replicated on every node participating in the call.

3.5.3 *MPI_Scatter()*, *MPI_Gather()*, and Related Functions

MPI provides a number of variations on standard scatter and gather routines. A scatter routine gets data that is present in a single process and spreads it out evenly among other processes. A gather routine is the reverse of a scatter routine in that it gets data spread out among a number of processes and collects it all at one process. Both operations are demonstrated in Figure 3.2. MPI provides two further routines. The first is *MPI_Allgather()* which performs a gather operation, except instead of returning all the values to one process, the values are returned to all processes so that they all end up with an identical set of data that was made up of the data from each individual process (this is shown in Figure 3.3). The other is *MPI_Alltoall()*, which is an extension of *MPI_Allgather()* to the case where each process sends distinct data to each of the receivers. The jth block sent from process i is received by process j and is placed in the ith block of that process' receive buffer. The operation performed by *MPI_Alltoall()* is shown in Figure 3.4.

Aside from these functions, "vector" variants of each provide additional flexibility. The vector variant of the gather routine (*MPI_Gatherv()*) provides an array parameter instead of

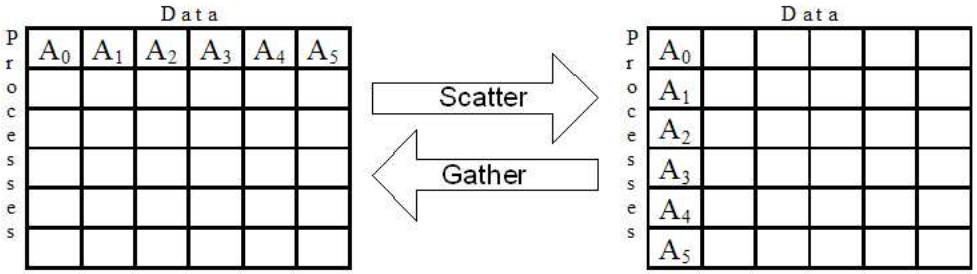

Figure 3.2: Scatter/Gather Example

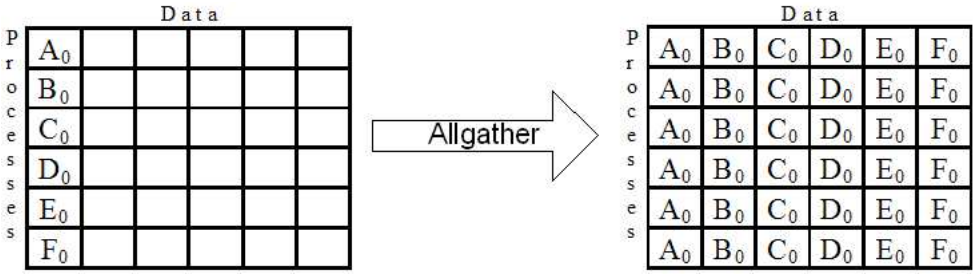

Figure 3.3: Allgather Example

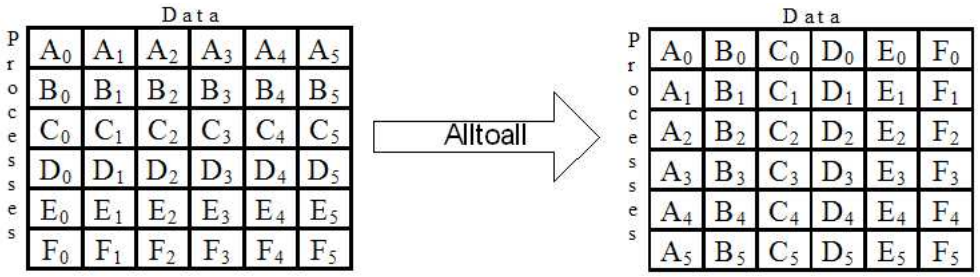

Figure 3.4: All-to-all Example

a scalar parameter for specifying the number of elements to gather from each process. This allows a different number of elements to be gathered from each process instead of the same number from each. It also provides an array of displacements that can be used to specify where the data should be stored in the buffer on the root process. The vector variant of the scatter routine allows the user to specify a different number of elements to distribute to each process, and a list of displacements relative to the beginning of the source array to specify where the root should send data from to each process.

MPI_Allgather() and *MPI_Alltoall()* have similar vector variants called *MPI_Allgatherv()* and *MPI_Alltoallv()*. *MPI_Allgatherv()* is just an extension of *MPI_Gatherv()* where all members receive the data instead of just the root. *MPI_Alltoallv()* allows processes to specify different values for the number of elements to send to and receive from every other process. It also allows you to provide an array of displacements for the send buffer and an array of displacements for the receive buffer. Finally, there is *MPI_Alltoallw()*, which is even more generalized in that it allows everything *MPI_Alltoallv()* does, but it additionally allows the sending datatype and receiving datatype parameters to be arrays so that every block sent and received can have a different datatype. This function can be used to emulate scatter and gather functions with the functionality to provide different datatypes for each block, which is why there is no "Scatterw", "Gatherw", or "Allgatherw" functions.

3.5.4 *MPI_Reduce()*

A Reduce operation gets data from multiple sources and reduces it into a single value by means of some operation. A common operation is to sum the values on many processes and have the root node end up with the final sum. The MPI implementation of this operation takes a source buffer address, a destination buffer address (which is only relevant on the root), the number of elements, the datatype of those elements, an operation to perform (of type `MPI_Op`), and a communicator. MPI also provides *MPI_Allreduce()* which does the same

process, except returns the result to all of the participating processes.

Chapter 4

MPI Implementation on SCMP

The MPI implementation described in this paper is guided by a few goals that should be stated up-front. First, it does not implement 100% of the MPI 1.2 standard. The goal is to provide an MPI implementation that will run most MPI programs, and therefore rarely-used functions that are non-trivial to implement have been omitted.

Next, functions have been designed to be algorithmically efficient, but an emphasis has been placed on code clarity and ease of modification over optimization. For example, many functions call other MPI functions — inlining this functionality would remove the function call overhead, but would mean longer functions and repeated code, both of which reduce readability and maintainability.

Finally, support for Intercommunicators (as opposed to standard Intracommunicators) has been neglected. The framework is in place, but the actual support is not there. This is mostly due to a lack of perceived applications and because their usefulness was limited before MPI 2.0, which was not implemented in this project. If the MPI 2.0 specification is implemented in the future, support for Intercommunicators should be added.

4.1 Hardware Considerations

A number of factors need to be considered when creating a software implementation for a specific hardware platform. One of the main features of the SCMP system over traditional multiprocessors is the high-bandwidth/low-latency on-chip network. Since MPI is a communications library, this will be a significant consideration. However, there are some other qualities of the hardware architecture that guide design decisions as well.

4.1.1 Network

A number of unique features of the SCMP network play into design decisions for a communications library. The most obvious is that it has an exceptionally low latency, high bandwidth network. This means that computation cost can potentially be on the same order of magnitude as communication costs. Another feature is that there are certain inherent communications commands built into the ISA of the SCMP processor. Primitives exist for sending message headers, register values, blocks of memory using DMA, and terminating messages.

One of the problems encountered with the SCMP network is that it only supports two types of messages – thread and data. Thread messages can be used as control messages or to send small amounts of data, but the amount of data a thread message can carry is limited by the number of registers available to a thread context.

Data messages can send potentially large amounts of data, but the sender is required to specify a destination memory address for this data. This was not a problem when the project was started and most of the memory in programs was statically allocated, but since the introduction of *malloc()* functionality for dynamically allocating memory, it is unlikely that one processor will understand another processors' memory mapping well enough to

determine where data should be placed on the remote processor. This means that some degree of handshaking must occur so that the processors can agree on where the data for a specific communication should go. The low latency of the network can help mitigate this problem, but some considerations brought up in the Threads subsection, section 4.1.3, below can cause problems. For a communications library to be efficient on this architecture, it needs to know when to use each type of message.

4.1.2 Memory

Since all memory is on-chip, the SCMP system has lower memory latencies than would be expected from systems with off chip memory. However, chip area is limited, and as a result there is limited memory for each processor. Generally, it has been considered that between 2 and 8 MB of memory will be available to each processing element. While this is not a trivial amount of memory, it is orders of magnitude smaller than what would be expected on a cluster of workstations, which is one of the popular configurations for a message-passing system. A communications system should be able to be fairly lean on memory, but this is still something to be aware of.

The current implementation is approximately 120 kB of compiled code and under normal use its dynamic memory footprint should be expected to be smaller than the code size. The per-message overhead is designed to stay as small as possible, and structures like linked lists are used whenever possible instead of allocating large static arrays that are likely to waste space. That said, sending large messages of type `MPI_PACKED` could cause problems since that requires that both sides of the communication have a "packed" version of the data transmitted in addition to the original data. This could potentially cause a problem if large messages are sent in this manner.

4.1.3 Threads

One of the largest implementation issues concerns the SCMP threading model. To keep the system simple from both programming and hardware implementation standpoints, only a form of cooperative multitasking is available. Threads suspend only when they explicitly give up control of the processor, or if they are blocked on a network call. This makes the programming model easier since critical sections that don't make use of network calls can be atomic if they don't call *suspendThread()*. It also simplifies the hardware implementation since preemptive multitasking typically requires some sort of clock interrupt.

Recall that there are two types of network messages: THREAD messages and DATA messages. DATA messages are fine if the sender knows where the data needs to go, but some negotiation is required before that is known. As a result, THREAD messages are the obvious choice for the negotiation phase. Unfortunately, cooperative multitasking means that there can often be a long period of time between when a THREAD message is received and when the thread it creates has a chance to actually execute.

This can be mitigated to a degree. In loops where a process is waiting on a message to complete, *suspendThread()* is called repeatedly to allow threads created by THREAD messages to be executed. However, it would be better if communication could progress when it is not being waited on, so when the data is needed it could already be there. Sometimes this occurs; however, it is hard to assure it will happen. The key is to minimize the amount of handshaking that goes on for any given data exchange. Unfortunately, a certain amount of handshaking seems unavoidable. Of course, on such a low-latency network, the performance impact can be barely significant.

4.2 Implementation Details

4.2.1 Datatypes

The functionality MPI provides for user-defined datatypes is powerful, but it needs to be analyzed with regards to the capabilities of the hardware. In general, it allows one process to send arbitrary pieces of data from arbitrary locations in memory to another process which can get those pieces of data and put them wherever it desires as well. SCMP's network support certainly allows for creating a message from data distributed throughout memory. However, it would need to be sent to a contiguous buffer on the remote side since only the remote side knows what the layout should be on that processor (the data layout can be different on both sides as long as the number and types of data objects is the same).

It is possible that the sending processor could learn the recipient's data layout by communicating with it, but the additional negotiation would probably offset any gains with this approach. Since the recipient processor will otherwise have to unpack the data at the destination, it was determined that explicitly packing any nonstandard data before sending was worthwhile. When a sending procedure is called with a non-primitive datatype, it packs the data into a buffer and calls itself with the new buffer as the source buffer and with the datatype as `MPI_PACKED`. When a receiving procedure is called with a non-primitive datatype, it calls itself with a temporary receiving buffer and datatype `MPI_PACKED` and after that returns, it unpacks the data into the real buffer. This approach keeps the code simple and performance reasonably high. The only clear improvement would be to pack the data into the network message on the fly, which would be fairly complicated and may not have much of a performance improvement.

4.2.2 Send/Receive

The MPI standard contains a number of variants of basic send/receive primitives. Since the MPI standard is so complex, it was decided that code reuse should be maximized. As a result, blocking send and receive calls are implemented with non-blocking calls and a call to *MPI_Wait()*. Due to message recipients often not knowing in advance which process will be sending the message, it was decided that most processing should occur on the receiving side. All the information needed about a specific message is included in an MPI_Request object on each side participating in the transfer. There are four request queues associated with each communicator. Two are for use with collective communications, and will be discussed later. The remaining queues are the "postedReceives" and "postedSends" queues.

The postedRecieves queue is used when the receiving side calls the nonblocking receive function, *MPI_Irecv()*, without a sender having advertised a matching message to send (Figure 4.1). When this occurs, a MPI_Request object is created with the location of the receiving buffer, the size of the data it expects, the address of the receiving node, the address of the sending node (or MPI_ANY_SOURCE if it will accept messages from any source), the tag associated with the receive (or MPI_ANY_TAG if it will accept messages with any tag), the number of the communicator being used, and the datatype expected. This MPI_Request object is then added to the postedReceives queue.

The postedSends queue is used when a sender tries to send a message, and no matching receive has been posted in postedReceives (Figure 4.2). When *MPI_Isend()* is called on the sender, it sends a thread message to the receiver calling *_MPI_Post_send()*. This routine checks to see if a matching message request has already been posted in the postedReceives queue. If there is a request, this function simply sends a thread message back to the sender that calls *_MPI_Request_data()*. If no such request exists, it adds this request to the postedSends queue. When the receiver finally calls *MPI_Irecv()* and finds a matching message in postedSends, it sends a thread message to the sender that calls *_MPI_Request_data()*.

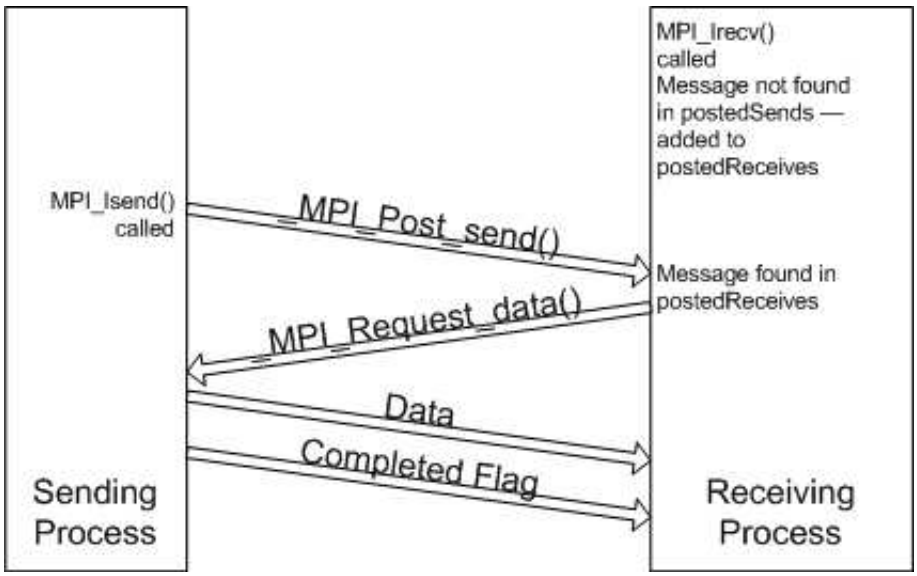

Figure 4.1: Receive-first Handshake

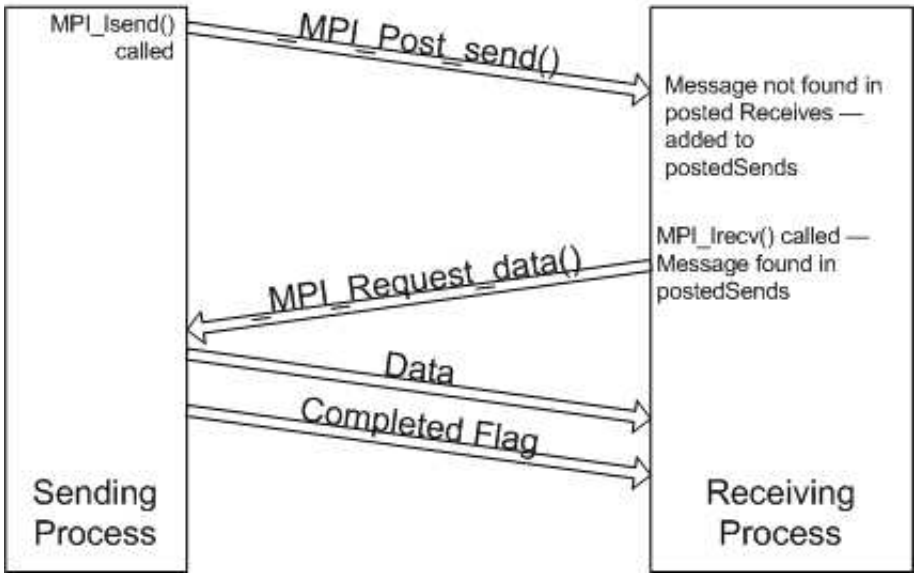

Figure 4.2: Send-first Handshake

When *_MPI_Request_data()* is called on the sending node, it is passed enough information to locate the MPI_Request object on the sending node, the memory address of the receiving buffer on the receiving node, and the memory address of the "completed" flag on the receiving side. This function simply looks in its local request object, finds the source buffer, sends that data in a data message to the receiving side's receiving buffer, and then sends a data message writing a '1' to the "completed" flag on the receiving side so it has a way of knowing that the data transmission is complete. It then sets the "completed" flag in its local MPI_Request object so the sending process knows that the send is complete.

Despite all of the messages involved, the latency of the send/receive process is surprisingly low. While there is certainly wasted time, when message queues are short and messages are of a reasonable size (hundreds of words or larger), the time required to send a message is fairly proportional to the length of the message, which implies that the overhead is tolerable.

One possible improvement would be to add a "short protocol" for very small messages that would remove some of the handshaking since data could be sent in a thread message. This would help minimize the latency of very short messages, which currently is very high. Also, hardware support for messages has been pursued for this architecture which could drastically reduce some of the latency problems.

A decision was made to avoid implementing the different modes of operation for sends. To start, ready mode sends would probably involve most of the same communication, and as such wouldn't be of much use. The standard mode send currently implements synchronous semantics, so that implementation is covered. As noted in [13], ready mode and standard sends may be implemented as synchronous sends without violating the MPI specification. The only failing is that buffered sends are not provided. In this author's opinion, the buffered semantics are only a substitute for proper use of non-blocking I/O, and as such can be omitted without impeding the power of the library. Also, in a limited memory environment, buffering doubles the memory footprint of larger messages, which is undesirable.

4.2.3 Communicator Functions

Communicators introduce a unique problem – these are identifiers that must be globally unique in a system. At least, if two processes have different communicators with the same identifiers, the groups associated with the two communicators must be disjoint, or else processes may get confused regarding which communicator a message is using. There are a number of possible solutions to this.

The first method developed involve processes using a reduction operation to perform a bitwise OR on an array of integers with each bit set to '1' if that communicator was in use on that process. The result would show which communicators were unused on all processes. It was eventually determined that, while this algorithm provides an efficient use of the space of communicator values, it is exceptionally slow since it requires a collective calculation.

Eventually, it was determined that the simplest solution was also the fastest. The highest numbered processor on the chip keeps a list of valid communicator values and whether they have been used or not. The last processor was used since the first processor takes control of a number of other calculations, and this way distributes the load somewhat better.

To duplicate a communicator (*MPI_Comm_dup()*), the node with rank 0 in the current communicator asks the last processor on the chip for a free context with a thread message. The first free context is sent back as a data message. Once this response is returned, the rank 0 node enters a barrier that the other nodes of the communicator have been waiting in. Once every node is in the barrier, they can exit the barrier, and every node but the rank 0 node sends a thread message to the rank 0 node asking for the new communicator context and the new context is returned to each node with a data message. The nodes use this information to duplicate the current communicator with a different context ID. The call ends in another barrier to ensure that no node tries to use the new communicator until everyone has it.

Creating a communicator (*MPI_Comm_create()*) works in a similar manner to duplicating a communicator, except that the process group for the new communicator is specified instead of just using the group from the current communicator. The *MPI_Comm_split()* function is much more complicated than *MPI_Comm_dup()*. The general process is the same, except all processes tell the rank 0 process their values for "color" and "rank", and the rank 0 process then calculates all of the communicators created, asking the highest numbered processor for free communicator contexts as it works. Finally, the rank 0 process distributes information about these new communicators to all callers of the function and a final barrier prevents any process from going forward and using the new communicators until all processes have the full information.

4.2.4 Collective Operations

While the basic point-to-point send and receive operations in MPI are likely to only add a fairly constant overhead to the existing SCMP operations, collective operations have an opportunity to be much worse. Most of the applications using native SCMP network instructions were designed by people who knew the architecture well enough to be able to optimize the communications patterns. Unfortunately, MPI abstracts away a good deal of the underlying architecture, so there is a potential for very poorly designed process layouts.

Ideally, the MPI implementation would find an optimal communications structure for every operation; however, that is very difficult. This implementation has opted for making collective communication functions that are reasonably efficient in most cases. Some special cases could be developed for certain circumstances where there is an especially efficient implementation, but in the interest of keeping the code manageable this has been ignored for the time being. As time goes on, the remaining bottlenecks of the implementation will be discovered, and extensive optimization in those cases can occur.

Barrier

Barriers are traditionally a bottleneck in parallel systems. In general, any sort of synchronization primitive tends to hurt performance since they require communication between a number of different processes.[14] In SCMP's libraries, there already exists a high-performance barrier routine; however, it doesn't have the flexibility to be used in the same contexts as the *MPI_Barrier()* function.

Since barrier calls in MPI are associated with a communicator, a variable `barrierCount` was added to the communicator data structure. This variable is initialized to 0 when the communicator is created. Whenever a process enters the barrier that isn't the rank 0 process in that communicator, it sends a thread message to the rank 0 process that increments the `barrierCount` variable for that communicator, then it waits for its `barrierCount` variable to be non-zero. When the rank 0 process enters, it increments `barrierCount` and waits for barrierCount to equal the size of the group, calling *suspendThread()* repeatedly to allow the thread messages time to execute. Once the rank 0 process sees that all processes in the communicator have entered the barrier, it resets `barrierCount` to 0, then it sends a thread message to every other process that sets the other processes' `barrierCount` variables to 1, which is the condition they are waiting on.

At this point, all processes are allowed to continue. It is possible this implementation would be better using data messages instead of thread messages for the final notification; however, since the processes are repeatedly suspending themselves while waiting, any latency from the use of thread messages should be minimal.

Broadcast

Due to the high-bandwidth of the SCMP network, a fairly straightforward approach to broadcasts seems to be reasonable. The current approach simply has every process that is not the root execute a blocking receive, and the root process executes a nonblocking send for every process in the communicator then waits on those messages to complete. As the number of processors increases, this procedure may start to become inefficient, so it is possible that in the future a better solution may be needed. The best option would probably involve looking at the size of the communicator and if it is large, use a tree-based distribution system. The current approach can have lower latency than the tree approach for small sets of processors, and that is why that implementation was chosen.

Scatter, Gather, and Variants

MPI_Scatter(), *MPI_Gather()*, and related functions all use similar techniques to *MPI_Bcast()*. Once again, for small numbers of processors, the technique of using point-to-point operations reduces the latency. If a tree structure is used to distribute data (or some other hierarchical communications topology), the second phase of communication cannot occur until the first phase has already occurred. This involves the full handshaking process between nodes, and the time required for the recipient to realize it has received the data. When sending point-to-point messages, the root can be responding to one message while another request is heading through the network, causing an effective pipelining of the message components.

Reduce

The reduction operation provided by the *MPI_Reduce()*, *MPI_Allreduce()*, and *MPI_Reduce_scatter()* functions is a special case. Since the operations used in the reduction procedure are required to be associative and are sometimes explicitly commutative[13], complex communications patterns can be used to reduce the data, and distributing the computation can compensate for the latency increases associated with hierarchical communications. Note that the scan operations (*MPI_Scan()* and *MPI_Exscan()*) must be chained together to complete correctly by having each process send the result it calculates to the next ranked process.

For the reduction operation, a simple algorithm is used that halves the number of processes participating in the communication every step, as shown in Figure 4.3. In the first step, every odd-ranked process sends its data to the even-ranked process with a rank one less than the odd process. From this point on, the odd processes are out of the calculation. In the next step, every process that is not a multiple of 4 passes data to the process numerically below it which is a multiple of 4. This process repeats until the only process left to pass data to is the rank 0 process, which is considered to be a multiple of every number. There are n steps in the process, where n is the smallest value for which $2^n >= communicator\_size$. So a communicator with 16 members would take 4 steps, and a communicator with 64 members would take 6 steps for a full reduction.

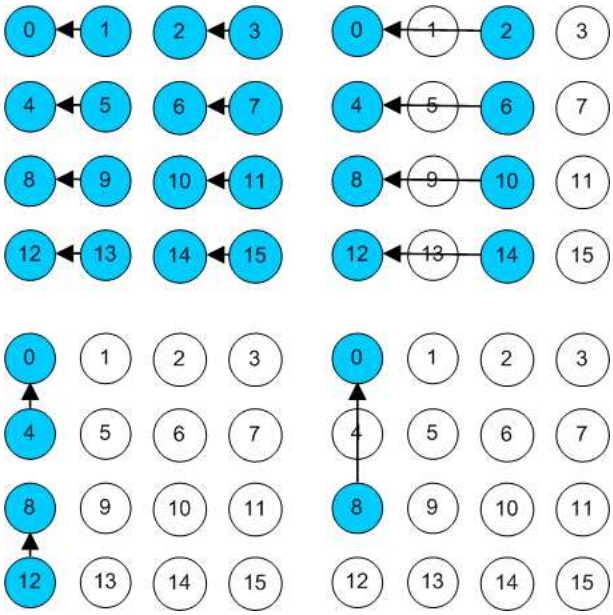

Figure 4.3: Communication steps for a reduction operation on a 4x4 processor grid

Chapter 5

Performance Analysis

The approach taken with regards to performance analysis of the MPI implementation is one of comparing the performance of existing SCMP benchmark programs with versions rewritten to use the MPI implementation. It is difficult to determine exactly what should be done to make the two versions comparable, so the general rule was to use the most obvious and simplest MPI commands for a given communication. By comparing MPI and non-MPI versions of the same program, the overhead of the MPI library can be analyzed in a convenient manner.

The two benchmarks chosen came from work done by William Wesley Dickenson [15]. Both are linear algebra matrix operations, which have traditionally been popular benchmarking tools for parallel systems (LINPACK is a prime example of a benchmark based on linear algebra). They both also have a strong communications component, which is important since the overhead of the MPI library would be diluted if applications with minimal communications were used. All benchmark times presented are in thousands of clock ticks (or microseconds, assuming a 1 GHz system clock). The times presented are only for the core of the algorithm. Initialization and finalization routines were omitted since they spend most

```
vectorX = zeroVector();
vectorR = vectorB;
vectorP = vectorB;
error = 1.0;
while(error > errorTolerance) {
   alpha = (transpose(vectorR) * vectorR) /
           (transpose(vectorP) * (matrixA * vectorP));
   nextVectorR = vectorR - alpha * (matrixA * vectorP);
   beta = (transpose(nextVectorR) * nextVectorR) /
          (transpose(vectorR) * vectorR);
   vectorX = vectorX + (alpha * vectorP);
   vectorP = nextVectorR + (beta * vectorP);
   vectorR = nextVectorR;
   error = |matrixA * vectorX - vectorB| / |vectorB|;
}
```

Figure 5.1: Pseudocode for Conjugate Gradient Benchmark

of their time reading and writing data, and currently the SCMP simulator does not attempt to accurately represent the time required for I/O operations.

5.1 Conjugate Gradient Benchmark

The Conjugate Gradient is commonly used to solve both linear and nonlinear systems of equations. In general, it solves the equation $Ax = b$ for x, where A is a sparse n x n matrix, b is a 1 x n vector, and x is an unknown 1 x n vector. The pseudocode for the algorithm is given in Figure 5.1.

In order to parallelize this algorithm for SCMP, rows of matrix A and portions of vectors b and r are distributed cyclically among the nodes. Each node has a copy of p and x vectors, but only manipulates its portion of p. For more detailed information about this benchmark and the SCMP implementation of it, see [15].

5.1.1 Characteristics

The main loop of this benchmark performs three reduce operations and $4 + n$ broadcast operations, where n is the number of processors. Out of those operations, all three reduce operations and four of the broadcast operations transmit only one `double` value, and transmitting small amounts of data is a fairly inefficient operation in this MPI library. However, the n remaining broadcast operations transmit a large amount of data ($num\_rows\_in\_A/num\_processors$ `double` values). This can be more efficient, however note that these broadcasts go to every processor, and collective operations on a large number of processors tend to have a lot of overhead.

5.1.2 Expectations

The collective calls with only one `double` value will be slow, but they are unlikely to dominate the time needed. The larger transfers are the important part, and the performance on those should be fairly competitive. For large data sets and small numbers of processors, the MPI implementation may actually do quite well.

5.1.3 Results

The performance results for the Conjugate Gradient benchmark are mixed (see Table 5.1 for performance numbers). As noticed in both benchmark programs used, there appears to be an overhead associated with increasing the number of processors that exceeds the overhead the native SCMP implementations experience. Also, larger data sets increase both the computation to communication ratio and increase the size of the communications, allowing the mostly-fixed overhead of the MPI calls to be amortized over a larger data transfer. While there are a number of runs that had over 100% overhead over the native SCMP version, also

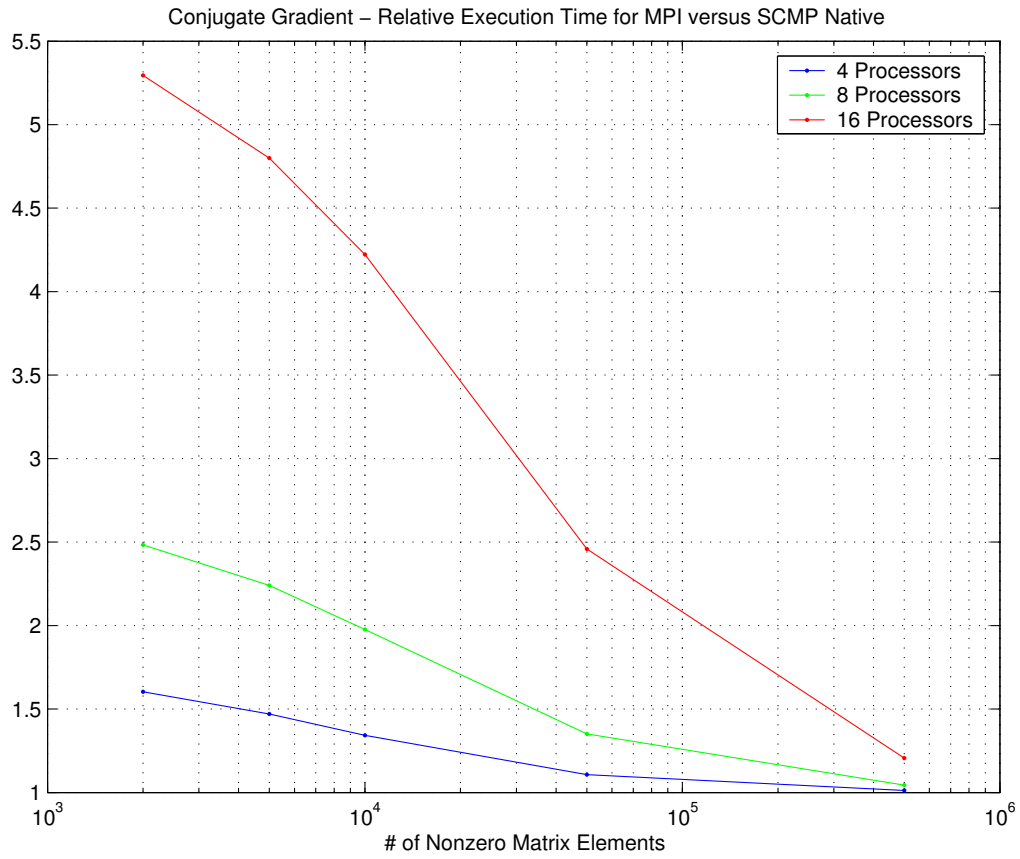

Figure 5.2: Relative Performance of MPI to Native SCMP Versions

notice that at 500,000 nonzero elements the runs all had fairly acceptable overheads. In Figure 5.2, it is clear that as the data size increases, the overhead diminishes rapidly. Even the 16 processor version, with 20.6% overhead, is potentially within tolerable limits.

It should be noted that accidently using a higher optimization setting on the compiler caused the 4 processor MPI version not only to beat the native SCMP version at 500,000 nonzero elements, but it actually was faster than the sequential solution divided by 4. Since the performance overhead of MPI in that situation can be more than compensated for by additional compiler optimization, that would probably be considered acceptible to most people.

Table 5.1: Conjugate Gradient Performance Numbers

# of Nonzero Elements	# of Processors	MPI Time	Native SCMP Time	Ratio
2000	4	2717	1695	1.603
5000	4	3203	2178	1.471
10000	4	4005	2983	1.343
50000	4	10462	9451	1.107
500000	4	82306	81293	1.012
2000	8	3146	1267	2.483
5000	8	3397	1516	2.241
10000	8	3797	1921	1.977
50000	8	6958	5150	1.351
500000	8	43064	41262	1.044
2000	16	5586	1055	5.295
5000	16	5712	1190	4.800
10000	16	5920	1402	4.223
50000	16	7493	3048	2.458
500000	16	25508	21150	1.206

0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
12	13	14	15	12	13	14	15
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
8	9	10	11	8	9	10	11
12	13	14	15	12	13	14	15

Figure 5.3: Processor Assignment to A

5.2 QR Benchmark

For a given matrix A, the QR decomposition calculates matrices Q and R such that $A = QR$, where Q is orthogonal and R is an upper-triangular matrix of the same dimensions as A. This calculation is used in a number of engineering applications such as signal processing and least squares calculations. This implementation uses the Householder method, which operates on each column of A in succession, transforming every element in that column below the diagonal into 0. After this has been done on all columns, the resulting matrix is R. For more detailed information about this benchmark and the SCMP implementation of it, see [15].

5.2.1 Characteristics

The assignment of processors to elements of matrix A is as if the processor array was tiled over the matrix. A processor i has the intersection of columns where $col\_num\%X\_Dim == i\%X\_Dim$ and rows where $row\_num\%Y\_Dim == i\%Y\_Dim$ (where X_Dim is the number of processors in the X-dimension and Y_Dim is the number of processors in the Y-dimension).

This is demonstrated with a mapping for a 4x4 grid of processors onto an 8x8 matrix in Figure 5.3.

For every column in the matrix A, first all processes with data from that column perform a reduce operation together. Then the owner of the diagonal element for that column sends it to the process with the top element in that column. Finally, the top element in the column broadcasts a `double` array and a single `double` value to the rest of the column.

After this is completed, every process enters a barrier. Note that processes not involved in that column were waiting in the barrier while the column completed its calculations. Then an array of `double` values and a single `double` value are both broadcast along the rows. Finally, a *MPI_Allreduce()* occurs along the columns. After this, the process repeats for the next column.

For analysis, it is assumed that in each iteration one column performs a reduce and two broadcasts, there is a global barrier, there are two broadcasts that every node performs along their row, and an allreduce (which is a reduce and a broadcast) is performed within each column.

5.2.2 Expectations

This benchmark is heavy on communication, especially collective communication. It uses a number of reduce and broadcast operations. This means that it will probably show a high performance penalty for using the slower MPI operations instead of the native SCMP ones. The worst case scenario will be with a large number of processors and a small amount of data per processor — the pathological case would involve a large processor array with one element of the matrix A per processor. That decreases the computation to communication ratio and would hurt the original system, but would cause even more performance degradation to a system like MPI with more communication overhead.

5.2.3 Results

The following results were found as shown in Table 5.2. This data includes cases that try to show the full range of performance that can be obtained from this library. First, consider the pathological cases. The worst is running the QR decomposition on an 8x8 grid of processors with an 8x8 matrix. This has a large number of processing elements, and each one only holds one data value. An operation like this is dominated by communication time. Notice that the MPI version runs over 7 times slower than the original version. That is the worst result out of every configuration attempted with this benchmark. Clearly, in this case, the MPI library adds a tremendous amount of overhead. There are a few other cases like this, and they occur whenever the amount of data per processor is low, which increases the communication to computation ratio.

There is also overhead created when the number of processors increases. Realize that computing the 256x256 matrix with 16 processors maps the same amount of data to each processor as the 128x128 matrix computed with 4 processors. While the 4 processor solution adds 29.3% overhead to the computation, the 16 processor solution adds 80.3% overhead. The same goes for using a 256x256 matrix with 4 processors (13.0% overhead) versus using a 512x512 matrix with 16 processors (40.2% overhead). This is further illustrated in Figure 5.4 and Figure 5.5 — in these cases, it is clear that the performance of the library falls off at high processor counts. The library does not scale well, and the blame for that most likely falls on the broadcast operation which is optimized for smaller processor configurations.

With the unpleasant results out of the way, there are also positive results present. The overhead goes as low as 2.9% for a 1024x1024 matrix with 4 processors. The 512x512 matrix with 4 processors has 6.0% overhead, which is also reasonable. In general, the performance of a 4-processor configuration is fairly good, as shown in Figure 5.6. The MPI version appears to be acceptable for large data sets, depending on the user's needs. Realize that sequential code on a modern microprocessor can perform this operation on a 1024x1024 matrix in a

Table 5.2: QR Decomposition Performance Numbers

Matrix Size	# of Processors	Native SCMP Time	MPI Time	Ratio
8x8	64	77	556	7.221
16x16	64	164	1181	7.201
128x128	4	17277	22340	1.293
128x128	16	5580	14205	2.546
128x128	32	3524	12479	3.541
128x128	64	2648	17576	6.637
256x256	4	126760	143258	1.130
256x256	16	35626	64229	1.803
256x256	32	19791	47988	2.425
256x256	64	11993	61566	5.133
512x512	4	974639	1033072	1.060
512x512	16	257151	360416	1.402
512x512	32	134709	233885	1.736
512x512	64	73206	248394	3.393
1024x1024	4	7652352	7872544	1.029

matter of seconds, so this problem is actually quite small in terms of problems that are usually run on a parallel system. Unfortunately, beyond 1024x1024 is very expensive to simulate, and there are also concerns of insufficient memory beyond that point (a `double` is 8 bytes, so on a 4 processor system, $1024 * 1024 * 8/4 = 2$ MB of data per processor out of a 8 MB maximum addressable space).

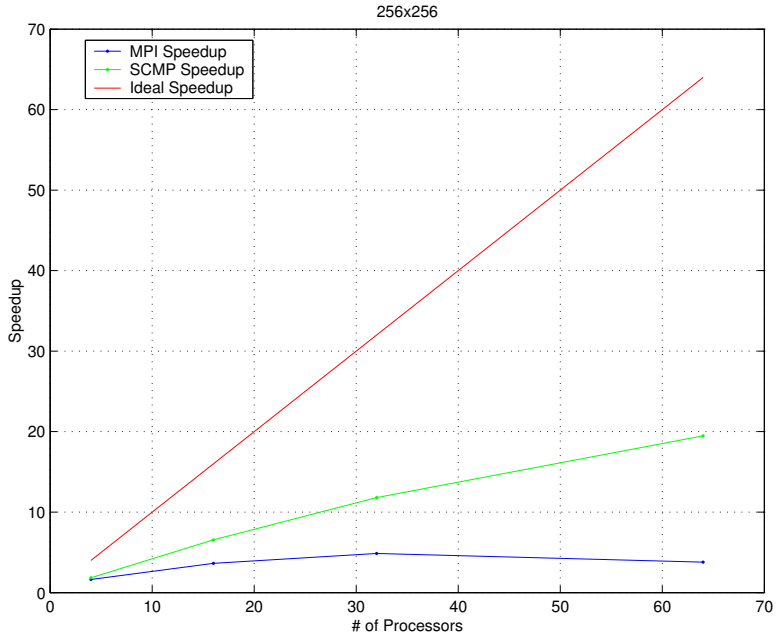

Figure 5.4: Speedup with a 256x256 Matrix

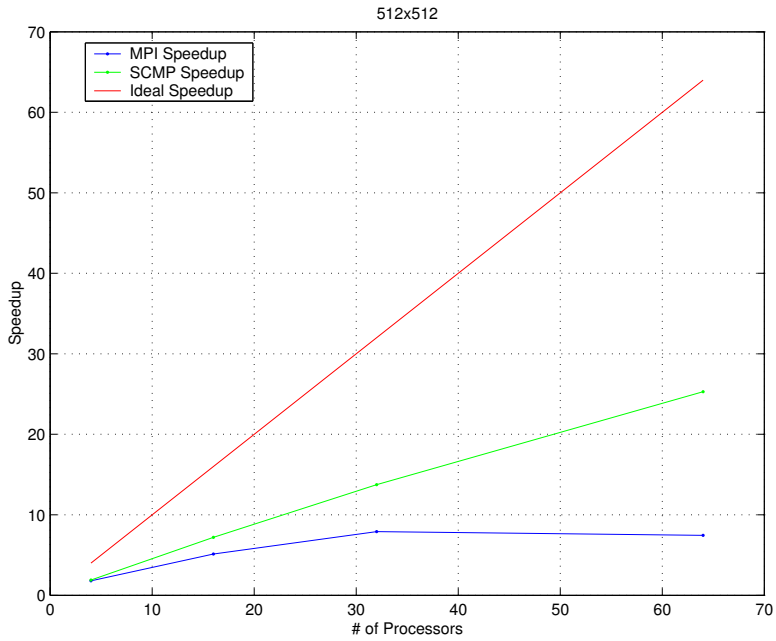

Figure 5.5: Speedup with a 512x512 Matrix

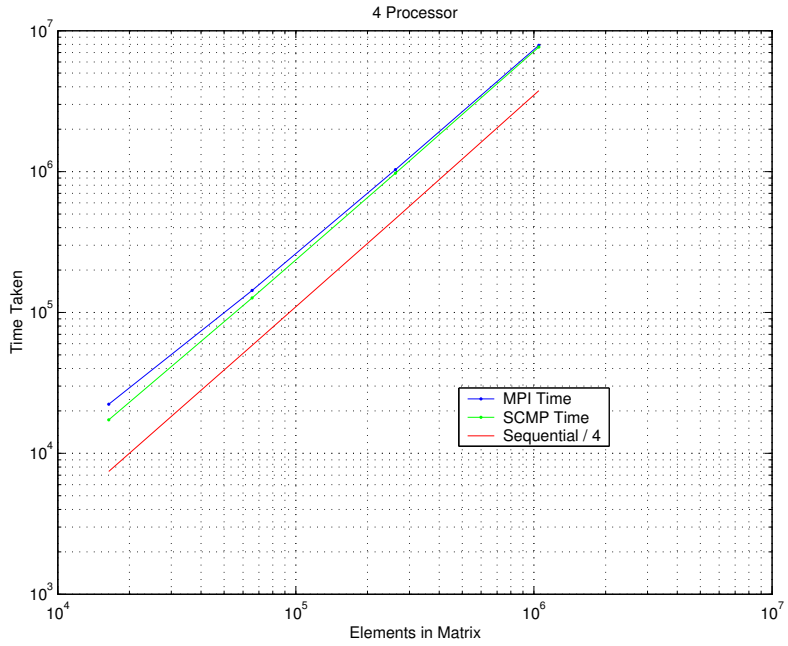

Figure 5.6: Execution Time for a 4 Processor System vs. Number of Nonzero Elements

Chapter 6

Conclusions

6.1 Observations on Findings

It is usually impossible to add a layer of abstraction to some process without hurting performance, and this MPI library proved no exception. Much of the functionality that MPI brings to the table was not used in the tests, so the code implementing that functionality ended up just being overhead. Also, these benchmarks were written with the normal SCMP communications model in mind, so they were designed to "push" data to remote nodes and had no need for a system that could perform a send before the other side needed to receive it.

The intention of those benchmarks was more to determine how much overhead the MPI library added to SCMP communications than to prove that efficient MPI programs could be written. As a result, these programs are not written as efficiently as they could be. Efforts were made to keep the communications semantics as close as possible to the original program to ensure that the performance numbers are representative of the MPI overhead instead of the skill of the programmer modifying the code to use MPI compared to the skill of the

original benchmark developer. The only changes that were allowed were when it was a natural translation; for example, many computations using SCMP THREAD messages were modified to use *MPI_Reduce()* as that is the natural replacement. Also, barrier calls were removed in areas where they were unneeded, since this implementation synchronizes on collective calls, whereas the previous methodology would have required explicit synchronization.

For example, in the Conjugate Gradient benchmark, the function *DistributeP()* is currently written as a loop that iterates through the processors. It has each processor build a contiguous buffer of the data to be sent, call *MPI_Bcast()* to broadcast it to the other processors, and finally the recipient processors unpack that buffer into their local data structure. This entire operation could be performed with one call to *MPI_Alltoall()* using a derived datatype. It is hard to say for sure if this would be faster, but this alternative would allow communication to be overlapped, most likely causing a significant performance improvement. It is possible that the increased network congestion could possibly make performance worse; however, that scenario is unlikely since a broadcast call already has a bottleneck at the node broadcasting the information.

The performance for applications with small data sets and/or a large number of processors was especially poor. This is a result attributable to a low computation/communication ratio. A variant of Amdahl's Law[16] seems to apply. In a general sense, Amdahl's Law states that if part of a computation takes $n\%$ of the time of the full computation, the best possible performance gain from optimizing that part of the computation is $n\%$. That is commonly used to show a theoretical limit on speedup when parallelizing a program, since part of any program is inherently serial and cannot be made faster with additional processors. In this case, if an MPI program spends, for example, 10% of its time in communication routines, it could not be sped up by more than 10% if the MPI routines were replaced by optimized native SCMP communications. Thus when the time spent in communication is small relative to the time spent in computation, it is unlikely that a large difference will be observed in the results from using native SCMP communication commands versus using the MPI implementation

presented in this thesis. This is shown in benchmarks run with a small number of processors on large data sets — in these cases, the overhead due to MPI is often less than the overhead of using a lower optimization setting on the compiler (for compiler performance statistics, see [17]).

An advantage of the SCMP hardware is fast, low-overhead communications. Aside from lowering the communications penalty of existing parallel applications, this feature also allows SCMP to attempt to undertake applications that previously were inefficient to parallelize due to frequent needs for communication between cooperating processes. While the MPI library is comparable to the native SCMP instructions when sending large blocks of data, when sending small amounts of data the overhead in the MPI library can make a significant negative impact on execution time. An important mitigating factor is that MPI was not designed for efficient fine-grained communications. MPI is designed for clusters and distributed memory multiprocessors, and the software that is written for such systems is aware that communications overhead is significant.

This leads the author of this thesis to reason that the MPI implementation presented herein is usable for most SCMP applications. There is significantly less code in the MPI versions of the benchmarks presented, and it can be inferred that the time required to write these benchmarks from scratch using MPI would be lower than the same task using the conventional SCMP communications instructions. When used in applications that are traditionally parallelizable, the performance degradation is minimal. For applications trying to take full advantage of the features of the SCMP network, it will still be worth the effort to write them using the native instructions instead of MPI. Every layer of abstraction has its cost/benefits tradeoffs, and MPI is no exception.

This implementation appears to be successful, as it seems to have met its two main goals. Its first goal was to provide a more user-friendly API for creating software for the SCMP system without hurting performance too much. Most applications will not see a dramatic

performance impact from using MPI and the smaller code size leads one to believe that development will be easier, especially if the user is familiar with MPI from other parallel systems. The other goal was to allow MPI programs from other systems to be easily ported to SCMP. Aside from a few incompatibilities mentioned in the next section, it appears that most MPI programs will run almost completely unmodified on the SCMP system. This will make the process of testing application performance much easier than having to write a SCMP version of every application for which SCMP performance results are desired.

6.2 Incompatibilities with Other Implementations

This implementation is designed to be as compatible with the MPI standard as possible, but there are a few things that need to be kept in mind when porting applications from other systems.

A change required in all MPI programs involves how the program is started. In a standard MPI system, a copy of the program is executed on every processor simultaneously. The SCMP system only starts executing the program on processor 0. This means that a program such as the following:

```
int main(int argc, char *argv) {
  int rank;
  int val;
  MPI_Init(argc, argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank==0) {
    val = 5;
    MPI_Send(&val, 1, MPI_INT, 1, 99, MPI_COMM_WORLD);
  } else if (rank==1) {
    MPI_Recv(&val, 1, MPI_INT, 0, 99, MPI_COMM_WORLD, MPI_STATUS_NULL);
    printf("Value received: %u\n", val);
  }
  MPI_Finalize();
  return(0);
}
```

Would need to be rewritten as:

```
#include "scmp.h"
void start(int argc, char *argv) {
  int rank;
  int val;
  MPI_Init(argc, argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank==0) {
    val = 5;
    MPI_Send(&val, 1, MPI_INT, 1, 99, MPI_COMM_WORLD);
  } else if (rank==1) {
    MPI_Recv(&val, 1, MPI_INT, 0, 99, MPI_COMM_WORLD, MPI_STATUS_NULL);
    printf("Value received: %u\n", val);
  }
  MPI_Finalize();
}
int main(int argc, char *argv) {
  parExecute(getXDim()*getYDim(), start, argc, argv);
  return(0);
}
```

Note the introduction of a *parExecute()* call to cause the main function (renamed *start()*) to be executed on all processors. Note that *MPI_Init()* must be called on *all* processors

in order for the library to work correctly. Since most implementations expect that *main()* will execute on all processors simultaneously upon starting the program, the safest way to modify the code is to make a *main()* function that only calls *parExecute()* to execute the old *main()* in parallel on all processors, as shown above.

A few MPI 1.2 features are not fully supported in this implementation. No support is provided for buffered sends, so if they are used in a program, the person porting the application would have to check an make sure that the program did not rely on buffering to prevent deadlocks. Also, no support exists for setting the mechanism by which errors are handled — they are always returned as the result of the calling function. Occasionally, if the implementation considers an error especially fatal, the execution may be stopped by an *assert()* call. In general, this only occurs when the processor would otherwise throw an exception later in the function and cause the program to abort anyway, so the *assert()* call provides additional clarity as to the source of the error. This implementation also does not support intercommunicators or communication topologies; however, both features appear to be rarely used in MPI programs.

For performance reasons, it would be wise to keep in mind that derived datatypes are handled by using *MPI_Pack()* and *MPI_Unpack()*. As a result no performance benefit exists in using derived datatypes versus using those functions. It would probably make sense to explicitly use *MPI_Pack()* and *MPI_Unpack()* instead of derived datatypes anywhere derived datatypes are not a perfect fit.

6.3 Summary of Work

The first step in creating this MPI implementation was to read through the MPI standard and determine how the standard could be mapped to the capabilities of the SCMP hardware. MPI functionality was then implemented in an incremental manner, with each function

tested as it was created. The resulting code was developed using design principles with a focus on code reuse and ease of modification. That methodology allowed subsystems of the MPI implementation to be completely rewritten at times when it became clear they were insufficient when adding new functionality.

After the library was complete, testing was performed with full MPI applications. This approach is compared to the tests performed during development that tested individual functions without regard to how they are used in a real application. During this time, bugs were found and corrected, and previously overlooked functionality was added to the library.

During the testing process, it became clear that the existing SCMP simulator left much to be desired in terms of debugging high-level libraries, such this MPI implementation. As a result, a new simulator was created that sacrificed cycle-accuracy for a rich array of debugging features. These features included bounds checking on all internal data structures, the usage of linker output to map function names to addresses, and the ability to detect MPI calls and retrieve data from the parameters for use in the simulator. An annotated screenshot of the simulator appears in Figure 6.1.

The bounds checking assisted in finding errors in the code. Some errors would cause the old simulator to modify data structures that caused symptoms in places far from the cause. For example, one processor modified the thread contexts of another processor due to a bad pointer. That kind of error is very difficult to trace since the symptoms show up in a processor unrelated to the problem, and this more secure implementation of the simulator caught problems before they were too far removed from the source.

Using the linker output enabled a number of useful features. Instead of just knowing the location of the instruction pointer, this simulator could show the entire call stack using actual function names. In the old simulator, if something inside a *printf()* call caused the processor to throw an exception, the user could tell that it occurred in a *printf()* call by looking at the instruction pointer when the exception was thrown. However, that function may be called

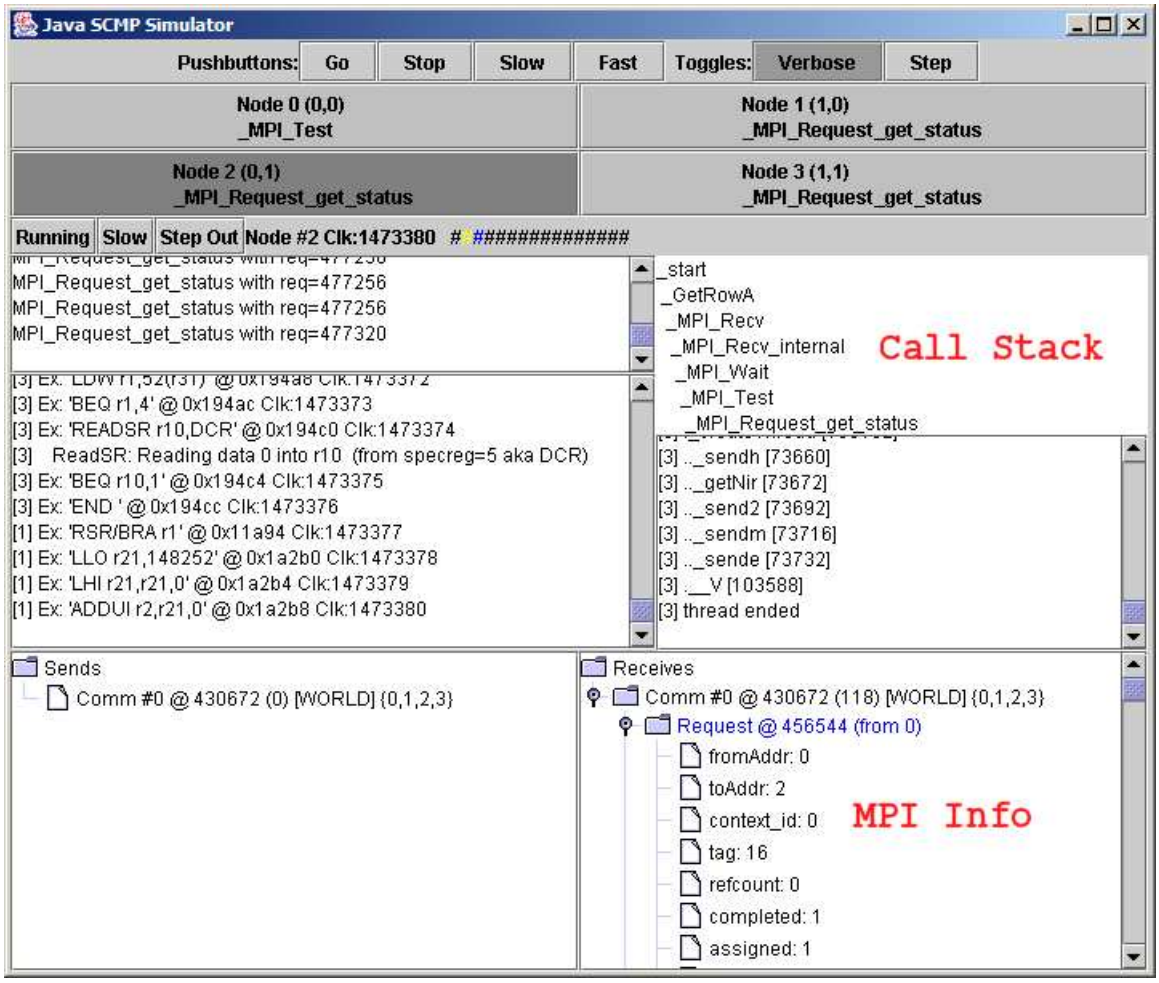

Figure 6.1: Screenshot of New SCMP Simulator

many times in a program, and it may not be immediately obvious where the error occurred. By showing the entire call stack, problems could be localized much more quickly. Having the address to function name mappings available also allowed the simulator to produce the parameters a specific function was called with, if desired.

Finally, the ability to interpret MPI calls was invaluable while debugging the MPI implementation. The simulator could show the status of the message queues associated with each communicator during the execution of the program, and that made it much more convenient to detect errors in message queue management and to determine the status of a communica-

tion at a given part of the program. This is a far cry from a simulator that provides register values and disassembled machine code.

Once the testing process was complete, some basic profiling was performed on the MPI applications. This helped focus optimization of the worst parts of the MPI implementation. Optimizations were only performed if they did not have a significant impact on the readability or maintainability of the code. A highly optimized version would require a number of modifications, many of which are referenced in the next section.

6.4 Future Work

Additional work could still be done with this implementation. An obvious area is in the realm of optimization. One source of overhead still present in this library is function call overhead. To keep the code readable, functions were reused whenever possible. As a result, calling one MPI function may result in a number of function calls. An example is *MPI_Allreduce()* — which calls *MPI_Reduce()* and *MPI_Bcast()* in order to perform its duties. However, *MPI_Reduce()* and *MPI_Bcast()* both call *MPI_Isend()* and *MPI_Irecv()*, and both of those functions call a helper function developed to simplify the creation of `MPI_Request` objects. Considering that the processors in the SCMP system can send 2 flits, or 64 bits of data, per clock cycle, the function call overhead is reasonably significant. An advanced compiler could inline a number of these functions; however, the SCMP compiler is currently not that sophisticated and may never be.

Additional experimentation is required to determine what algorithms are fastest for various MPI functions. For example, in some cases a broadcast is best performed by simply having the root send the data to each process individually. However, as the number of members in the communicator increases, this becomes less efficient. A high-performance implementation of MPI on this system would probably have multiple techniques for performing a broadcast

and pick the best one based on the situation at hand. Such an implementation may also be able to make use of the layout of the processors participating in a collective call to optimize the communication. If MPI becomes the communications library of choice for SCMP, this would be time well-spent.

Another optimization would involve creating a "short protocol" for data transfers. If a send operation was started in order to send a few words of data, that data could easily be packed into a THREAD message in order to eliminate the overhead associated with handshaking to pass the data. If this was done, it is possible the thread created on the receiving end may need to create an intermediate buffer for the data and copy it to the real destination when a matching receive call is made.

There are a few MPI functions that were not implemented in this version of the library, and it would be advantageous to have a 100% compatible MPI implementation so these extra functions could be added. The addition of intercommunicators would not be especially difficult and could make some programs with hierarchical communications patterns easier to design. Also, the additions in the MPI-2 standard have been mostly ignored. While not all of the MPI-2 features seem to make sense for SCMP, some of them could be useful.

A natural next step for this MPI implementation would be to integrate it with the work on hardware support for send/receive based messages described in [11]. If this modification alone only removed a few cycles from the time to perform any given communications operation, a noticeable performance improvement would result. One would have to make sure the hardware support matched MPI's needs closely, or else the improved performance may be overshadowed by extra code to work around the differences.

If the SCMP architecture were to be redesigned for MPI, there are a few changes that could be made. The most obvious is removing the limitation of cooperative multitasking. In fact, the only time MPI uses thread contexts is in order to handle message traffic, not for running separate threads. If thread messages could preempt the main thread, the efficiency of the

MPI library could be dramatically increased. Of course, if this was the case, there would need to be a way to isolate critical sections in the main code. If threads were serviced quickly, the system could have many less thread contexts since they currently only serve to buffer incoming thread messages. If preemption was not allowed, more thread contexts could help since there is a noticeable performance degradation when any operation causes more thread messages to get sent to a node than it has free contexts to handle.

Finally, the next obvious step is to start using the MPI library. Original programs could be created for SCMP using the MPI functions, and existing programs could be easily ported to the SCMP system. With the current system, porting a standard benchmark efficiently and correctly to SCMP can take months of time. Having a working MPI implementation could lower that to weeks or perhaps even days, dramatically improving the productivity of SCMP researchers.

Bibliography

[1] James M. Baker Jr., Sidney Bennett, Mark Bucciero, Brian Gold, and Rajneesh Mahajan. SCMP: A single-chip message-passing parallel computer. In *PDPTA*, pages 1485–1491, 2002. 1

[2] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *PDPTA*, pages 282–293, 2000. 1

[3] Bob Colwell. Things CPU architects need to think about, Feb 2004. URL: `http://stanford-online.stanford.edu/courses/ee380/040218-ee380-100.asx`. 2

[4] John L. Hennessy. The future of systems research. *IEEE Computer*, 32(8):27–33, Aug 1999. 2

[5] University of Tennessee. *MPI: A Message-Passing Interface Standard*, 1.1 edition, June 1995. 3, 4

[6] IMPI Steering Committee. *IMPI: Interoperable Message-Passing Interface*, 0.0 edition, January 2000. URL: `http://impi.nist.gov/`. 3

[7] T.V. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, 1992. 8

[8] G.A. Geist and V.S. Sunderam. The evolution of the PVM concurrent computing system. In *Compcon Spring '93, Digest of Papers*, pages 549–557, 1993. 9

[9] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259. ACM Press, 2000. 11

[10] Brian Gold. Balancing performance, area, and power in an on-chip network. Master's thesis, Virginia Polytechnic Institute, 2003. 11

[11] Charles W. Lewis Jr. Support for send and receive based message-passing for the single-chip message-passing architecture. Master's thesis, Virginia Polytechnic Institute, 2004. 19, 70

[12] William Gropp and Ewing Lusk. *User's Guide for MPICH, a Portable Implementation of MPI*, 1.2.1 edition, 1996. http:// www.mcs.anl.gov/mpi/mpiuserguide/paper.html. 19

[13] Marc Snir and Steve Otto. *MPI — The Complete Reference: Volume 1, the MPI Core.* MIT Press, 1998. 43, 48

[14] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach (Third Edition).* Morgan Kaufmann Publishers, 2003. 46

[15] William Wesley Dickenson. High-performance applications for the single-chip message-passing parallel computer. Master's thesis, Virginia Polytechnic Institute, 2004. 50, 51, 55

[16] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967. 62

[17] Sidney Bennett. Designing a compiler for a distributed memory parallel computing system. Master's thesis, Virginia Polytechnic Institute, 2003. 63

Vita

Jeffrey Hyatt Poole was born on November 12, 1981 in Miami, Florida. In 1999, he graduated from South Lakes High School in Reston, Virginia. He went on to Virginia Polytechnic Institute and State University for a Bachelor of Science in Computer Engineering, finishing in December 2002.

Overlapping with his Bachelor degree, Jeffrey started his Masters of Science in Computer Engineering also at Virginia Tech in the Fall of 2002 and will complete the degree requirements in July 2004. In August 2004, he will begin working for L-3 Communications, a developer of military communications hardware, as part of the Network Architecture group in Salt Lake City, Utah.