Support for Send-and-Receive Based Message-Passing for the Single-Chip Message-Passing Architecture

Charles W. Lewis Jr.

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

Master of Science
in
Computer Engineering

Dr. James M. Baker Jr., Chair
Dr. Mark T. Jones
Dr. James D. Arthur

April 28, 2004
Blacksburg, Virginia

Keywords: Parallel Computing, Message-Passing, Single-Chip Computer
Copyright 2004, Charles W. Lewis Jr.

Support for Send-and-Receive Based Message-Passing for the Single-Chip Message-Passing Architecture

Charles W. Lewis Jr.

James M. Baker, PhD, Committee Chair

Department of Electrical and Computer Engineering

Abstract

Arguably, from the programmer's perspective, the programming model is the most important characteristic of any computer system. Perhaps this explains why, after many decades of research, architects and programmers alike continue to debate the appropriate programming model for parallel computers. Though thousands of programming models have been developed, standards such as PVM and MPI have made send-and-receive based message-passing the most popular programming model for distributed memory architectures. This thesis explores modifying the Single-Chip Message-Passing (SCMP) architecture to more efficiently support send-and-receive based message-passing. The proposed system is compared, for performance and programmability, to the active messaging programming model currently used by SCMP.

SCMP offers a unique platform for send-and-receive based message-passing. The SCMP design incorporates multiple multi-threaded processors, memory, and a network onto a single chip. This integration reduces the penalties of thread switching, memory access, and inter-process communication typically seen on more traditional distributed memory parallel machines. The mechanisms proposed in this thesis to support send-and-receive based message-passing on SCMP attempt to preserve and exploit these features as much as possible.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

What will computer systems look like in ten or twenty years? This question drives the research of computer architects everywhere. Similarly, computer programmers ponder how the computer systems of ten to twenty years from now will be programmed. While some small amount of debate exists as to which of these two questions is more important, they are, in fact, both critical to the development of a successful computer architecture. After all, a computer architecture is useless if it cannot be effectively programmed to meet the computational needs of its users.

System designers must consider hardware and programming model characteristics in tandem. The two components are equally important to the success of the end product and they are thoroughly interdependent. Therefore, as a preface to discussion of modifications to the Single-Chip Message-Passing (SCMP) architecture to support a new programming model, the first chapter of this thesis provides an overview of programming models and the current SCMP architecture.

1.1 Programming Models

As computers have grown in power, they have become increasingly complex. While the earliest computers were programmed in binary code, the complexity of today's machines make this an unreasonable chore. To ease the programmer's task, modern computer systems provide several layers of abstraction between the programmer and the hardware. These include assembly languages, high level programming languages, and software libraries. The representation of the computer system that is the end result of these layers of abstraction is called the programming model.

Two general classes of programming models for parallel systems exist. Shared memory programming models present a single address space to all processes. One process may receive data from another by reading from the same address that the other process wrote to. Conversely, distributed memory programming models divide memory into discrete units and allow only one process to access each unit. Under distributed memory models, processes typically communicate by exchanging messages through some communication medium such as an Ethernet network. Distributed memory programming models that provide interprocess communication through messages form a subgroup called message-passing programming models. As its name suggests, the SCMP architecture fundamentally supports message-passing programming models.

Currently, SCMP provides programmers with a modified version of the Active Messages [24] programming model developed at Berkeley by Thorsten von Eicken and colleagues. Von Eicken created the Active Messages model to increase the overlap of communication and computation. Under this model, every message carries the address of a handler to invoke on the receiving node. These routines, written by the application programmer, are responsible for extracting the data from the network and integrating it into existing threads of computation. A detailed description of the original message-passing system for SCMP is presented in Chapter 3.

Thanks in part to industry-wide standards such as the Message Passing Interface [14], or MPI, and the Parallel Virtual Machine [23], or PVM, send-and-receive based message-passing programming models are more popular today than Active Messages. Under send-and-receive based message-passing, users explicitly control message transfers. For a message transfer to complete, a send operation on the source node must be matched to a receive operation on the destination node. The programming model does not specify how the transfer is completed; it only requires that matching send and receive operations execute on the source node and the destination node respectively.

1.2 An SCMP Overview

The past two and a half decades have seen substantial growth in computer performance. To fuel this growth, computer architects have leveraged the reduction of transistor sizes to enable their chips with higher clock speeds and increased support for Instruction Level Parallelism, or ILP. However, some researchers believe that computer architects will not be able to continue on this path much longer. Therefore, many researchers have begun redirecting transistor resources in hopes of finding new means of obtaining performance gains. The SCMP architecture, in development at Virginia Tech under the guidance of Dr. James Baker [3], is an experimental architecture that directs its transistor resources toward parallelism among processes and threads. Threads are instructions streams that can access one another's address spaces, and processes are instruction streams that cannot access one another's address spaces.

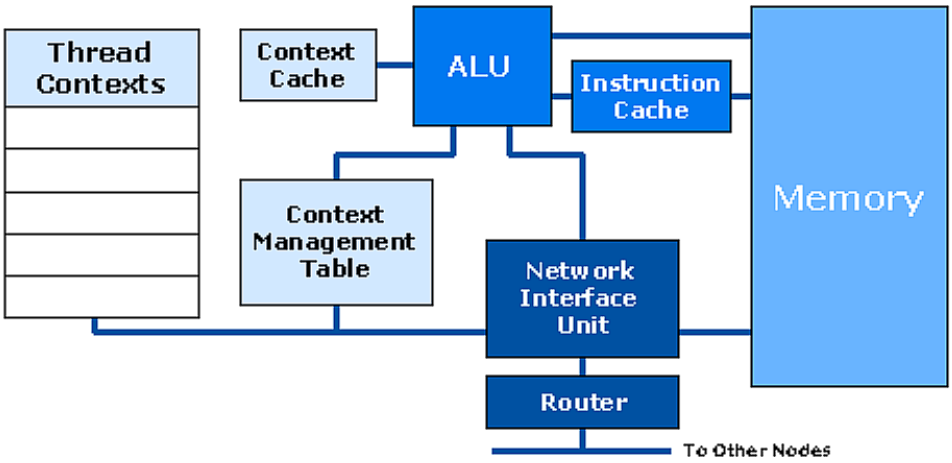


Figure 1.1: Block Diagram of an SCMP Node.

1.2.1 SCMP Nodes

Thread Level Parallelism, or TLP, is supported on SCMP at the node level. Each processing node contains a simple 32-bit RISC pipeline, up to 8MB of memory, a network interface unit, and a router. Each processing node also supports up to 16 thread contexts of 32 registers each. Figure 1.1 shows a block diagram for the SCMP processing node. Threads are given control of the pipeline according to a preemptive round-robin scheduler implemented in hardware. By using a separate register set for each thread, and hardware-based scheduling, thread switches can be accomplished in as few as 3 or 4 clock cycles [3]. This level of fast thread switching allows for fine grained TLP without suffering a significant penalty.

1.2.2 The SCMP Network

To support Process Level Parallelism, or PLP, SCMP connects many processors together on a single chip. As shown in Figure 1.2, SCMP is a tiled architecture, with the processor nodes arranged in a two-dimensional mesh. To avoid long cross-chip wires, processing nodes are

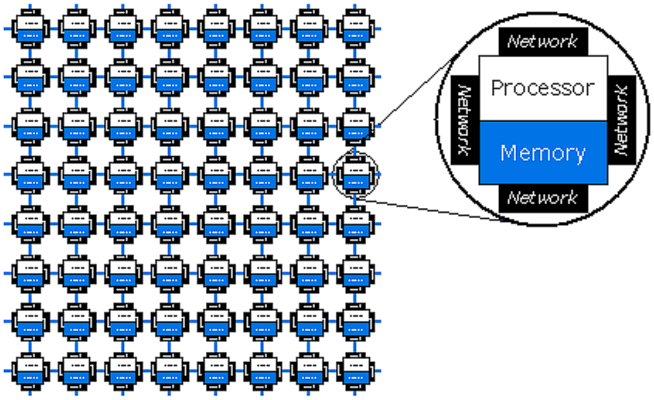


Figure 1.2: SCMP system with 64 Tiles.

only connected to their four nearest neighbors. The network created by these wires and the routers to which they are connected allows for cross-chip communication. This connection mechanism ensures that no wire is longer than a single node.

Messages are sent between SCMP nodes as streams of 32-bit flow-control digits, otherwise known as flits. The SCMP network uses wormhole routing. Under wormhole routing, only the head flit of a message contains any routing information. As it moves through the network, it reserves a path for the remaining flits of the message. The tail flit of the message releases the path as it traverses the network. This form of routing allows a router to forward a flit on to the next router before it has received the entire message. To route the head flit, the SCMP network uses dimension-order routing. This routing algorithm has been shown to be deadlock free as long as all flits entering the network eventually leave the network [10].

1.2.3 Shortcomings in the Original SCMP Programming Model

SCMP nodes transfer data to other nodes in data messages. Under the original message-passing system, data messages contain the address where the data should be stored at the

destination node. This feature allows the Network Interface Unit, or NIU, to quickly store data messages as they arrive, but does not allow threads to control the flow of data into the node. To regulate data transmission, SCMP programs under the original message-passing system use a request and reply programming model. Under this programming model, a node that is ready to receive data will initiate a thread on the node that holds that data. The new, remote thread will send the data to the requesting node and terminate.

As discussed further in Chapter 3, this programming model has several flaws. It works best when the address of the data at the replying node is known, which is not always possible when dynamic memory is used. When a node holds data needed by many other nodes, it can become inundated with threads requesting data. Finally, the programming model uses a second message to notify the requesting node that the data has arrived. By using a second message for synchronization, the programming model creates a race condition between the data and synchronization message.

1.3 Thesis Overview

This thesis will discuss modifications to the SCMP architecture designed to support two modes of send-and-receive based message-passing. These modes are based upon the portable MPI implementation named MPI Channel Interface [11], or MPICH. This thesis will discuss the justification for send-and-receive based message-passing support, a modified SCMP design, and a performance analysis of the new architecture.

The remainder of this thesis is organized as follows. Chapter 2 presents background information about messaging systems, including several successful architectures that provide message-passing communication. Chapter 3 provides a description of the SCMP architecture and programming model as it exists today. Chapter 4 gives an overview of concepts related to send-and-receive based message-passing. The next chapter, Chapter 5, proposes a mes-

saging system implementation to support these features on SCMP. A performance analysis of the modified SCMP architecture is presented in Chapter 6. The final chapter, Chapter 7, summarizes the findings of this thesis and outlines future work in this area.

Chapter 2

Background and Related Work

2.1 Messaging Systems

Implementations of message-passing programming models rely on messaging systems to carry out message-passing communication. Messaging systems consist of a thin software layer known as the messaging layer, a network interface adapter, and the network itself. According to Duato *et al.*, messaging layers have seven critical characteristics that affect the correctness and performance of message-passing communication. These are packetization, network interface control, user/kernel interface, copying and buffering, in-order message delivery, reliable message delivery, and protection [8]. However, for the sake of performance, many of the responsibilities of the messaging layer may be implemented in the network or the network interface. Therefore, for the purposes of this discussion, these characteristics will be attributed to the entire messaging system. Additionally, notification of communication events is an important responsibility of the messaging system.

2.1.1 Packetization

Typically, a user process that wishes to send a message supplies the messaging system with information about the destination process, the message payload, and sometimes a message identifier. It is the job of the messaging system to collect user-supplied message information, produce any additional data required for message delivery, and arrange the data according to the network's message format. This procedure is known as packetization. The term packetization is unfortunate, however, because it suggests that messages are sent as a number of individually routable packets as in TCP/IP networks. In fact, messages may be sent as packets, as streams, or as a whole [8].

2.1.2 Network Interface Control

Moving data and control information from the user process to the network interface is an important design consideration for the messaging system. The manner in which the messaging system handles this communication can greatly affect message-passing communication performance. However, the node architecture dictates the communication methods available from user processes to the network interface. Unfortunately, these methods are often unfavorable for high performance message-passing networks. Therefore, improving or circumventing the native process to network interface communication mechanism is a primary goal of many messaging systems [8].

2.1.3 User/Kernel Interface

Many message-passing multi-computers run an operating system. On such systems, whether the message handlers of the messaging layer run in user space, or from the kernel, significantly impacts performance. Early messaging layer implementations executed message handlers in

the kernel. Such designs required an expensive context switch into the kernel for each message handler invocation. Additionally, these early designs read messages into system buffers before copying them into the user's data space, increasing message latency. Therefore, most modern messaging layers cut out the operating system kernel by providing direct communication from user processes to the network interface and implementing the message handlers in user space [8].

2.1.4 Copying and Buffering

For a network routing protocol to remain free of deadlock, all messages sent to a destination must eventually be consumed. Deadlock freedom proofs generally satisfy this requirement by assuming infinite memory at the sender, receiver, or both. A receiver with infinite memory could immediately retrieve and buffer all arriving messages. A sender with infinite memory could buffer every out-going message for retransmission. This would allow the receiver to discard any message it could not immediately handle. However, real systems have limited memory. Therefore, some flow control mechanism must exist in the messaging layer to ensure that buffer space can be reused without deadlock or loss of messages.

The buffering policy implemented by the messaging system affects communication performance in terms of latency and sustainable bandwidth. Some buffering policies copy messages in and out of several buffers before they reach a data structure in the destination process. Each copy operation adds time to the message latency. Additionally, the communication bandwidth is limited by the slowest buffer stage and the smallest buffer pool. The destination can only retrieve messages from the network as fast as its slowest buffering stage. If messages arrive at a sustained rate greater than the slowest buffering stage can accept them, they will consume all the buffers before that stage. When the destination can no longer accept new messages, it must retard the transmission rate of the sender through flow control. Some flow control mechanisms require additional message traffic prior to message

transmission to guarantee buffer space at the destination. These flow control mechanisms also increase latency [8].

2.1.5 In-Order Delivery of Fragments

Depending on the maximum transmission size of the network, the messaging system may segment large messages into smaller fragments. These fragments must arrive at the destination process reassembled and in the correct order. Some networks guarantee in-order delivery of message fragments. If the network does not do that, however, higher levels of the messaging system must correct the order of fragments before passing the message to the destination process. TCP/IP sequence numbers are an example of a reordering mechanism [8].

2.1.6 Reliable Message Delivery

Unlike many Internet applications, parallel programs require reliable communication. The messaging system must guarantee that all messages sent will eventually be received by their destination process. The network, the network interface, and the basic message-passing software may all take part in ensuring message delivery. For example, many high-performance networks guarantee delivery of individual transmission units. However, even on systems with such robust networks, message delivery may fail if the network interface or message-passing software do not provide adequate buffering and flow control [8].

2.1.7 Protection

Message protection supports reliable delivery. If it is possible for messages to intersect in the network, they must be uniquely identified to ensure messages arrive at the appropri-

ate destination. Keeping messages isolated from one another is known as message protection. Similarly, in a multi-programmed environment, processes must not interfere with the message-passing state of other messages. If message handling occurs in the kernel, message protection can be achieved using traditional operating system memory protection schemes. If message handling occurs in user processes, then message protection can be provided through other schemes. For example, the same identification mechanism used for protection in the network can be used among processes if they can be trusted to cooperate [8].

2.1.8 Event Notification

Upon completion of any communication operation, such as a send or receive, it is the responsibility of the messaging system to notify the correct user process. Notification methods include interrupts and flags. Alternatively, the messaging layer may provide user processes direct access, such as polling, to message states maintained by the messaging system.

2.2 Message Passing Machines

SCMP differs from most message-passing machines in two very important ways. The majority of multi-computer nodes incorporate the network adapter as a peripheral device. Therefore, communication with the network adapter must traverse the host's external I/O bus which is typically orders of magnitude slower than the host processor. Because it is a custom single-chip system, SCMP overcomes this problem. An SCMP node's processor, memory unit, and network interface are all on the same piece of silicon. Communication among these subsystems occurs at the processor's clock rate. Additionally, many message-passing parallel systems provide an operating system. In such systems, incorporating or circumventing the operating system with the messaging system is a major design consideration. This is not an

issue for the SCMP parallel computer since it currently has no operating system. Should an operating system be developed for SCMP in the future, it can be designed to match the messaging system, instead of matching the messaging system to it.

Despite these differences, however, traditional message-passing machines offer valuable insight for SCMP. Today's message-passing systems are the result of decades of research in message-passing communication. This chapter discusses four popular message-passing systems on the market at the time of this thesis. The messaging systems of these machines each contain elements that may be used to support send-and-receive based message-passing for SCMP.

2.2.1 The IBM SP2

The IBM SP2 was one of the earliest machines to implement an optimized messaging system. A high-performance switch and host adapters form the core of the SP2 system. IBM offers several alternatives for nodes, though most configurations include a Micro Channel I/O bus [15]. The messaging system is completed by the messaging layer "known under two equally unimaginative names: External User Interface (EUI) and Message-Passing Library (MPL)" [22].

To avoid processor overhead, the MPL executes in the user's address space. The MPL consists of two layers: an API layer, and a pipe layer. The API accepts data from the user process and packets it for the pipe layer. The pipe layer maintains a bidirectional pipe for every communication partner. By providing a separate pipe for every communication partner, messages are protected at the host. While in the network, messages are protected by uniquely identifying them based on the communication endpoint and a message type field.

Pipes are reliable, buffered, and ordered byte streams. The pipe layer uses acknowledgments for reliable delivery, and a token protocol to ensure sufficient buffering is available at desti-

nation processes. Data is transferred between the pipe buffers and the network by copying it into reserved DMA buffers in kernel memory. The pipe layer communicates control information with the network controller using programmed I/O. The API layer allows the process to poll the pipe layer for event notification.

2.2.2 Myrinet

Because it grew out of networking research related to the groundbreaking Mosaic project [13], Myrinet [4] is a local area network targeted specifically for the multi-computer environment. The physical Myrinet network is made up of any number of pipelined crossbar switches and host adapters. The network uses blocking cut-through routing wherein once the routing header of a packet has been received, it can be forwarded onto the out-going channel before the rest of the packet arrives. This eliminates the store-and-forward latency a message experiences as it traverses some number of hops across the network. Myrinet transmits messages in streams; therefore, there is no need to ensure in-order delivery of fragments. Myrinet does, however, ensure reliable delivery across at least the network hardware.

GM for Myrinet

Myricom, the commercial developer of Myrinet, also provides GM, a Myrinet based messaging system [16]. GM provides isolated communication on a shared network in much the same way as TCP/IP. Processes may open any number of GM ports for sending and receiving messages. As messages travel across the network, they are differentiated by the node ID and port number of the source and destination nodes. When a process opens a GM port, it implicitly claims a number of send and receive buffers located in the memory of the network adapter, called LANai.

Like the MPL pipe layer, the GM interface manages send and receive buffer space by implicit tokens. Tokens represent space allocated in internal GM queues located in LANai memory. At initialization, each thread owns a number of send and receive tokens. Unlike MPL, send and receive operations are not blocked when token are depleted. Instead, a send or receive operation will have undefined results if the process does not have enough tokens. Therefore, it is the responsibility of the user process, or an intervening library, to manage tokens. By calling a GM send function a thread implicitly relinquishes a send token. The token is returned when the send completes. Similarly, a thread cannot receive a message of a particular size and priority until it has provided GM with a receive token of corresponding size and priority. The receive token specifies the buffer in which to store the matching receive. Once the matching message has been received, GM returns the receive token. Because the network guarantees reliable delivery between adapters, ensuring buffer space through token management is sufficient for guaranteeing full delivery.

GM only runs on systems that allow user processes to allocate DMA memory regions within their own address space. This allows GM to transfer message data between user buffers and LANai buffers without an additional copy to reserved DMA memory. GM communicates control information to the adapter by writing to LANai memory across the I/O bus directly. Completion of send operations are signaled through calls to user specified callback functions. However, users must poll for completion of receive operations.

Illinois Fast Messages for Myrinet

Illinois Fast Messages provides a much more streamlined messaging system for Myrinet [17]. Borrowing from the Active Messages model [24], Fast Messages transfers most of the message handling responsibility to the user. As in the Active Message model, each message contains the address of a user defined handler to invoke at the destination upon message delivery. These handlers are responsible for retrieving the message from the messaging system and

managing it thereafter. Additionally, these handlers inherently notify the process of receive operation events.

By requiring user processes to retrieve and protect their own messages, buffer management is much simpler for Fast Messages. The messaging system does not need to maintain a separate buffer pool for each communication pair. For Myrinet, Fast Messages uses only four buffers: LANai send, LANai receive, host send, and host reject. There is no host receive buffer as it is the message handler's responsibility to retrieve messages from the LANai receive buffer. Outbound messages, however, are copied into the host send buffer to wait until space appears in the LANai send buffer. Only the host reject buffer addresses reliable delivery of messages. Fast Messages uses an optimistic delivery strategy. Messages are sent immediately, without checking for destination buffer space. If the destination cannot accept a packet, it returns it to the sender. Returned messages are placed in the reject buffer for retransmission.

2.2.3 InfiniBand

While Fast Messages for Myrinet seeks to transfer most of the responsibility for handling messages to the user process, InfiniBand takes the opposite approach [12]. InfiniBand moves as much of the messaging layer functionality to the network interface as possible. Under most messaging system designs, the messaging layer notifies the network interface that a message needs to be sent by transferring the message to it. However, the InfiniBand architecture only requires the messaging layer to instruct the network interface to send a message located at an address in user space. The network interface then retrieves the message directly from the user process's buffer and transmits it when the destination is ready. Similarly, the messaging layer instructs the network where to place incoming messages when they arrive. This method creates a one-copy communication system. Messages are copied across the network directly user buffer to user buffer. To ensure messages are not sent before the destination is ready for them, the sending adapter must request and receive permission from the destination before

transmission begins.

Communication instructions are managed in InfiniBand in work queues. Each communication channel is defined by at least one pair of queues: a send queue and a receive queue. Queue elements are implemented in-order. Therefore, if a process cannot determine the order in which two or more messages will arrive, it must place the corresponding receive queue elements in separate queues. To notify processes of completion events, one or more work queues are associated with a completion queue. Completion queue elements contain all the information needed to identify the work queue element that created them.

The network interface retrieves messages from DMA memory regions in the user process's address space. If a message is too large to traverse the network in whole, the network interface will segment the message into packets. The network interface attaches routing and sequence numbers to the packets to ensure in-order delivery of packets. To further guarantee reliable delivery, the interface can be configured to require an acknowledgment for each packet.

2.2.4 Gigabit Ethernet

Gigabit Ethernet is the latest incarnation of the widely installed Ethernet networking protocol [19]. To capitalize on the parallel communication potential of Gigabit Ethernet, researchers have developed a non-buffering messaging system, Ethernet Message Passing (EMP), which borrows from Myrinet and InfiniBand [21]. Like the InfiniBand messaging system, the EMP messaging layer only instructs the network interface to collect and send data from DMA regions in the user process's address space. The network interface is responsible for packetization, reordering fragments, and ensuring message delivery. To request transmission of a message, the messaging layer sends a message descriptor to the network interface's memory through programmed I/O. The processes may then poll the network interface to receive completion notification.

Like Fast Messages for Myrinet, EMP sends messages immediately, with no mechanism to guarantee they can be accepted at the destination. Unlike the Fast Messages system, under EMP a destination node drops messages it cannot handle instead of returning them. When a destination node successfully retrieves a message from the network, it sends the source an acknowledgment. The network interface will resend the message from the sending process's buffer until an acknowledgment is received. A user process polling the status of a send operation will not see completion until the source has received an acknowledgment. The sending process must not modify its send buffer until it observes that the operation has completed.

Chapter 3

The SCMP Architecture

3.1 Architectural Trends

According to the well respected International Technology Roadmap for Semiconductors [20], or ITRS, computer manufacturers will be capable of fabricating over a billion transistors on a single chip by 2007. It is not clear, however, how computer architects will effectively utilize resources of such quantity. To wield billions of sub-micron transistors, computer architects must overcome several challenges that they have not faced before.

To fit over a billion transistors onto a single chip, the dimensions of every chip feature must shrink. This includes the cross-sectional area of wires used for communication among chip components. As the cross-sectional area of a wire decreases, however, its resistance increases. Consequently, signals propagate through the wire more slowly. Compounding the problem, both clock rates and the relative distance between modules typically increase with transistor count. This problem has led researchers to estimate that, by the time the billion-transistor threshold is crossed, less than 1% of a chip will be accessible in a single clock cycle [1].

To produce the performance gains of the past decade and a half, computer architects have increasingly devoted chip resources to exploiting parallelism among single instructions. As an architectural tool, this form of parallelism, known as Instruction Level Parallelism, presents many challenges. In order to extract extremely high levels of ILP, architectural designs must be very complex. Many designs include modules for fetching multiple instruction from the cache simultaneously, for executing instructions out of order, and for predicting the results of branch instructions before their arguments have been evaluated. This complexity requires long periods of design and verification. It also makes modules larger and farther apart on the chip, thus exposing designs to the wire latency problem. Most importantly, however, ILP faces diminishing returns. In general, computer applications contain a finite amount of ILP. Modern designs will soon be able to exploit nearly all the ILP present in most applications [7].

SCMP is one possible solution to the problems associated with progressive use of ILP. Instead of incorporating complicated modules to extract large amounts of ILP, SCMP targets TLP and PLP through a simple replication scheme. SCMP chips contain many simple and identical processors called SCMP nodes. Because it replicates simple nodes across the chip, the SCMP architecture should be easier to design and verify than more complex uni-processor architectures.

3.2 The SCMP Communication Network

To avoid long communication wires that cause wire latency problems, SCMP nodes are only connected to their four nearest neighbors. These connections form a two-dimensional mesh network through which SCMP nodes communicate. The network is controlled by two hardware modules on each node, namely the router and Network Interface Unit (NIU).

SCMP nodes compose messages as streams of 34-bit flits. As shown in Figure 3.1, flits contain a head flag, tail flag, and a variable payload. SCMP routers transport flits across

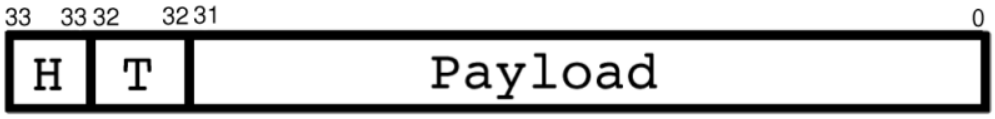
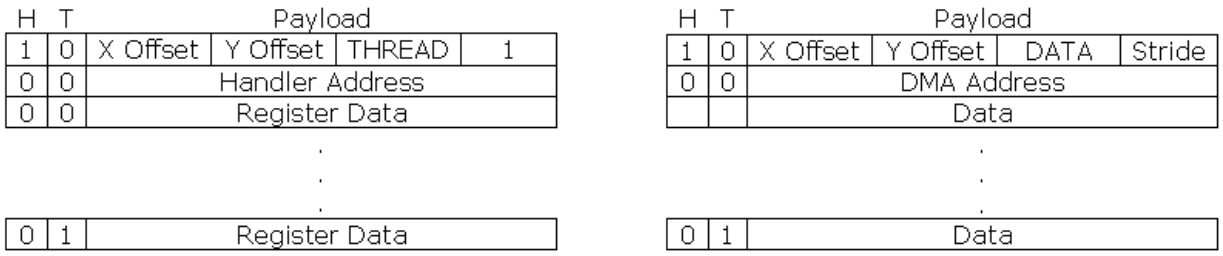


Figure 3.1: General SCMP Flit Format

the network according to wormhole and dimension-order routing. Only the head flit of a message contains routing information. Routing information is specified in the head flit as an X and Y grid offset pair from the source node to the destination node. Using dimension order routing, the SCMP routers move the head flit along the X axis of the processor grid and adjust the X offset with each hop. Once the X offset reaches zero, the flit has reached its destination column. The process then continues along the Y axis.

SCMP routers contain up to five bi-directional communication channels. Four of these channels connect the router to the node's neighbors to the north, south, east, and west. The fifth channel, known as the injection/ejection channel, connects the router to the node. Because the SCMP network is a two-dimensional grid, routers on the edge of the network will have fewer channels. Across these physical communication channels, each router multiplexes a number of virtual channels. Virtual channels are sets of buffers large enough to hold a handful of flits. These virtual buffers prevent a flit stream from taking full control of a physical channel which may cause deadlock [10].

As the head flit moves through the network, each router reserves a virtual channel for the flit stream on the next router in the message's path. The router maintains rights to this virtual channel until it transmits the tail flit to the next router. This path of virtual channels is the basis for wormhole routing. After the head flit passes through a router, that router knows where to place all remaining flits of the message. Therefore, non-head flits do not contain routing information. Additionally, wormhole routing eliminates store-and-forward delays seen in packet switched networks. Because next hop information is maintained between flit transmissions, a router can transmit the current flit without waiting for the entire message



(a) Thread Message Format

(b) Data Message Format

Figure 3.2: Original SCMP Message Formats

to arrive.

To interact with the network, each SCMP node contains an NIU. The NIU relieves the pipeline from most of the messaging management. Incoming messages can be stored in either a thread context or in memory, depending upon their type, without intervention from the pipeline. Outgoing messages, however, are built by instructions issued from the pipeline. The pipeline can offload memory transfers to the NIU by one of two memory transfer instructions described in the next section.

3.3 The SCMP Adaptation of Active Messages

The original message-passing system design for SCMP is based on an active-messages style found on the J-Machine [6] and PICA [25] systems. The protocol provides two message types: thread messages and data messages shown in Figure 3.2. Thread messages create threads of execution on the destination node. Data messages transfer data to the destination node.

Because this message passing system is intended to require as little intervention from the main processing unit as possible, each of these message types contain all of the information required by the NIU to process them. Because they contain their own handling information,

SCMP messages borrow from concepts proposed by Thorsten von Eicken and colleagues in the Active Messages [24] programming model. The second flit in each SCMP message format contains an address. When receiving a thread message, the NIU fills a free entry in the Context Management Table (CMT) and uses the address in the second flit as the new thread's initial instruction pointer. When receiving a data message, the NIU uses the address as the base pointer for a DMA transfer into memory.

Because the NIU must create a new thread before it can complete the reception of a thread message, thread messages may block in the NIU ejection channel if the node has no free thread contexts. If all of the virtual channels multiplexed across the NIU ejection channel are consumed by blocked thread messages, then no data messages will be able to reach the NIU. It is possible, however, that all of the threads on the node cannot finish until they receive some data message. Therefore, none of the threads will complete, and the NIU will be unable to drain the thread messages from the network, thereby creating deadlock. To prevent this scenario, SCMP routers partition their virtual channels into sets. A given set may only be used by a single message type.

SCMP messages are built in the pipeline using the instructions shown in Table 3.1. The pipeline must begin a message with a `sendh` instruction in which it specifies the destination node, destination address, a stride at which data should be separated at the destination, and the message type. For thread messages, the destination stride always equals one. The message body may be filled using any combination of `send`, `send2`, and `sendm` instructions. Instructions `sende`, `send2e`, and `sendme` are identical to their mnemonic counterparts except the last flit they generate will have the tail flag raised to mark the end of the message.

Notice that no instructions exist to receive data. The pipeline takes no role in receiving data. Data arrives outside the pipeline's knowledge and control. Therefore, threads must manage data arrival through higher level programming model constructs created by the messaging library shown in Tables 3.2 and 3.3. To keep data arrival synchronized with the

Table 3.1: SCMP Assembly Level Messaging Instructions

Opcode	Arguments	Description
`sendh`	d_node, type, d_address, d_stride	send a header flit
`send`	data	send one data flit
`send2`	data, data	send two data flits
`sende`	data	send one data flit and end message
`send2e`	data, data	send two data flits and end message
`sendm`	l_address, {l_stride & count}	send data block from memory
`sendme`	l_address, {l_stride & count}	send data block from memory and end message

data's destination thread, the data's source node will not transmit data to the destination until it receives a request. Because data arrival occurs without the pipeline's knowledge, the data's source node must notify threads on the data's destination node that is has completed the data transmission by sending a second message to raise a synchronization flag in the destination node's memory.

3.4 Problems with Active Messages for SCMP

Consider the simple data transfer between nodes A and B in Figure 3.3, wherein node A requires data stored on node B. To initiate the data transfer, node A must create a thread on node B which will send the data to node A. The new thread on node B must know the ID of the requesting node, the destination address, the destination stride, the source address, the source stride, and the number of values to send. Node A can provide the new thread with its node ID, the destination address, the destination stride, and the number of values to send in the thread message. However, it may not know the source address and stride if

Table 3.2: SCMP Active-Messaging Library – Thread Operations

Operation	Arguments	Description
createThread	int dst_node	create a thread on dst_node
	void(*addr)()	
	void(*callback)()	
	...	
parExecute	int num_nodes	create a thread on num_nodes nodes
	void(*addr)()	
	...	
getBlock	unsigned int node_id	request a block of values from node
	char *dst_addr	node_id
	unsigned int dst_stride	
	char **src_addr	
	unsigned int src_offset	
	unsigned int src_stride	
	unsigned int num_words	

Table 3.3: SCMP Active-Messaging Library – Send Operations

Operation	Arguments	Description
sendDataIntValue	int dst_node int *dst_addr int value	send an integer value to dst_node
sendDataFloatValue	int dst_node double *dst_addr double value	send a floating-point value to dst_node
sendDataBlock	int dst_node int *dst_addr int dst_stride int *src_addr int src_stride int count	send a block of values count words long from memory to dst_node (blocking)
sendDataBlockNB	int dst_node int *dst_addr int dst_stride int *src_addr int src_stride int count	send a block of values count words long from memory to dst_node (non-blocking)

dynamic memory allocation is used.

Two solutions to this problem exist under the original SCMP programming model. If the programmer wishes to use the general request function getBlock, listed in Table 3.2, the program must contain a global pointer to the requested block. Regardless of the address of the message block, the global pointer to it will reside at the same address on every node. Node A may then communicate which block it would like to request by sending the address of the global pointer in the thread message. Alternatively, the programmer may write a separate handler for each block that may be requested. These handlers contain hard-coded values for the source address and stride. While these methods are functional, neither lends itself to an eloquent programming style. A data exchange system that abstracts the data source and destination address is preferable.

The second stage of the transfer in Figure 3.3 illustrates a more serious shortcoming of the current SCMP message-passing system. Because the main processor is not involved in the reception of data messages, and the message-passing system provides no event mechanisms, there is no way to notify threads on the receiving node that data has arrived using only a single data message. Therefore, all data transfers must include a second data message. This second data message carries a single value of 1, and its address flit contains the address of a synchronization flag on the receiving node. Unfortunately, this system has the potential to introduce errors in SCMP execution. SCMP provides no guarantees for in-order transfer of messages between two nodes. Therefore, it is possible for the synchronization message to arrive at node A before the data message it signals. If a process on node A sees the flag raised before that data message has arrived, it will read unknown values from memory.

A final flaw in the Active Messages system for SCMP can cause a performance bottleneck. Each SCMP node can support up to sixteen threads in hardware at one time. However, an SCMP chip may contain up to 64 nodes. If a node, say node B, holds data needed by every other node on the chip, then it may become inundated with thread messages requesting

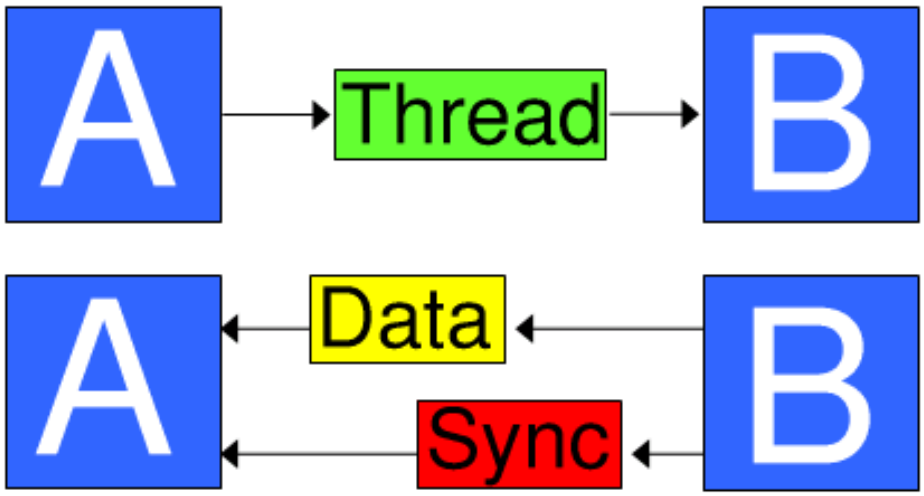


Figure 3.3: A Simple Rendezvous Data Transfer Between Two SCMP Nodes

data. After all of node B's thread contexts are consumed, thread messages will block at its NIU until some threads finish, or the exception handler can swap threads into memory. This scenario can significantly degrade performance.

Chapter 4

Send-and-Receive Based Messaging

4.1 Arguments for Send-and-Receive Based Messaging

Recent standard and portable programming models built on top of send-and-receive based message-passing systems have made this type of messaging layer very popular. Well know examples include MPI [14], and PVM [23]. While these robust programming models may or may not be suitable for SCMP, implementing even a very simple programming model based on a send-and-receive messaging layer can solve the problems associated with the active messages based programming model described in Chapter 3.

4.1.1 Local Data Flow Control

While many send-and-receive messaging layers offer the programmer more advanced features, they are all built around two primitive operations. The send operation transmits data to a receiving process. The receive operation accepts data from a sending process. For a message transfer to complete, the sending process must execute a send operation and the receiving

process must execute a receive operation for the same message. If a message arrives at a node before any process on that node executes a receive operation for that message, the node may buffer, block, discard, or return the message. However, it cannot place the message in the address space of any process. This characteristic of send-and-receive messaging-passing allows the receiving process to control the inbound flow of messages with local operations alone. Therefore, implementing such a messaging layer for SCMP would eliminate the need to generate threads on the data's source node to request a data transfer.

4.1.2 Abstract Message Identification

Because the receiving process must invoke an operation in the messaging layer to receive a message, it can provide the messaging layer additional information about that message as arguments in that operation. For example, receive operations generally require the receiving process to dictate the address at which to store the data from an inbound message. Consequently, the sending process no longer needs to know the address at which to store the message data at the destination because the message no longer needs to contain this information. To store the data from an inbound message, the destination node only needs a mechanism to associate the message with a receive operation executed by one of its threads. Abstract message identifiers are the simplest and most common mechanism used to associate messages with receive operations. Message identifiers may be as simple as an integer, or take the form of more complicated data structures. Using a messaging layer that supports abstract message identifiers on SCMP would eliminate the need for global buffer pointers or custom handlers for thread messages that request data.

4.1.3 Local Event Generation

Issuing a receive operation associates the receiving thread with the handling of an inbound message. The messaging layer uses this association to provide the receiving thread with notification of message arrival. The messaging layer may create a record of the receive operation with state information available to the receiving thread for polling. Alternatively, the receiving layer may raise a flag or execute a callback function in the receiving thread's address space to signal message completion. Many notification mechanisms are possible, most of which do not require additional message traffic. Therefore, implementing a send-and-receive based messaging layer on SCMP would remove the need for a separate synchronization message and eliminate the race condition it creates.

4.2 A New SCMP Programming Model

Changing to a send-and-receive based messaging layer system enables SCMP programmers to use a new programming model. This new programming model includes send and receive functions that most parallel programmers are more familiar with. However, it also retains functions to utilize SCMP's unique thread capabilities. In fact, as Chapter 5 will discuss in further detail, this new programming model does not affect SCMP's thread hardware or thread messages at all; it only pertains to communication of data. Programmers use this new programming model through library calls listed in Tables 4.1, 4.2, and 4.3.

Both the send and receive portions of the modified SCMP message-passing library include functions for blocking and non-blocking operations. Blocking operations do not return until the message-passing operation has completed, whereas non-blocking operations return immediately. A send or receive operation is considered complete when the programmer may safely access the message buffer used for the operation. For a receive operation, this means

Table 4.1: Modified SCMP Messaging Library – Thread Operations

Operation	Arguments	Description
createThread	int dst_node	create a thread on a dst_node
	void(*addr)()	
	void(*callback)()	
	...	
parExecute	int num_nodes	create a thread on num_nodes nodes
	void(*addr)()	
	...	
getBlock	unsigned int node_id	request a block of values from node
	int *dst_addr	node_id
	unsigned int dst_stride	
	char **src_addr	
	unsigned int src_offset	
	unsigned int src_stride	
	int message_id	
	unsigned int num_words	

Table 4.2: Modified SCMP Messaging Library – Send Operations

Operation	Arguments	Description
SCMPSendInt	int dst_node int message_id int value	send an integer value to dst_node
SCMPSendFloat	int dst_node int message_id double value	send a floating-point value to dst_node
SCMPSend	int dst_node int msg_id int stride int *address int count	send a block of values count words long from memory to dst_node (blocking)
SCMPSendNB	int dst_node int message_id int *address int stride int count	send a block of values count words long from memory to dst_node (non-blocking)
SCMPPollSend	int message_id	poll the status of a send operation
SCMPWaitSend	int message_id	wait for a send operation to complete
SCMPClearSend	int message_id	clear a send operation from the system

Table 4.3: Modified SCMP Messaging Library – Receive Operations

Operation	Arguments	Description
SCMPReceive	int message_id int *address int stride	receive a message and place the data at address (blocking)
SCMPReceiveNB	int message_id int *address int stride	receive a message and place the data at address (non-blocking)
SCMPPollReceive	int message_id	poll the status of a receive operation
SCMPWaitReceive	int message_id	wait for a receive operation to complete
SCMPClearReceive	int message_id	clear a receive operation from the system

that the data has arrived and resides in the message buffer. For a send operation, this means that the messaging layer no longer needs to read the message buffer to complete the data transfer. Therefore, the sending process may freely modify the buffer's contents.

Both the send and receive portions of the modified SCMP message-passing library also contain poll, wait, and clear functions. The poll functions allow the programmer to check the status of an message-passing operation, and react to the status however he or she sees fit. The wait functions similarly check the status of a message-passing operation, but always suspends the invoking thread if the operation is not complete. The clear functions allow the programmer to clear a message-passing operation from the messaging layer. The programmer must clear an operation before he or she issues another operation with the same message ID.

Chapter 5

Design

5.1 Message-Passing Modes

To support send-and-receive based message-passing for SCMP, two distinct message-passing modes were investigated. The two modes, a Ready mode and a Rendezvous mode, are based on message-passing modes found in the MPICH [11] implementation of MPI. The Ready mode uses an extremely light message exchange protocol, illustrated in Figure 5.1, to reduce message latency. Ready mode does not use any flow control protocol to ensure that the destination is ready to accept the data. If a message arrives at a node before any of the node's threads executes a receive for it, the node simply discards the message. Therefore,

Figure 5.1: A Ready Mode Transfer Between Two SCMP Nodes

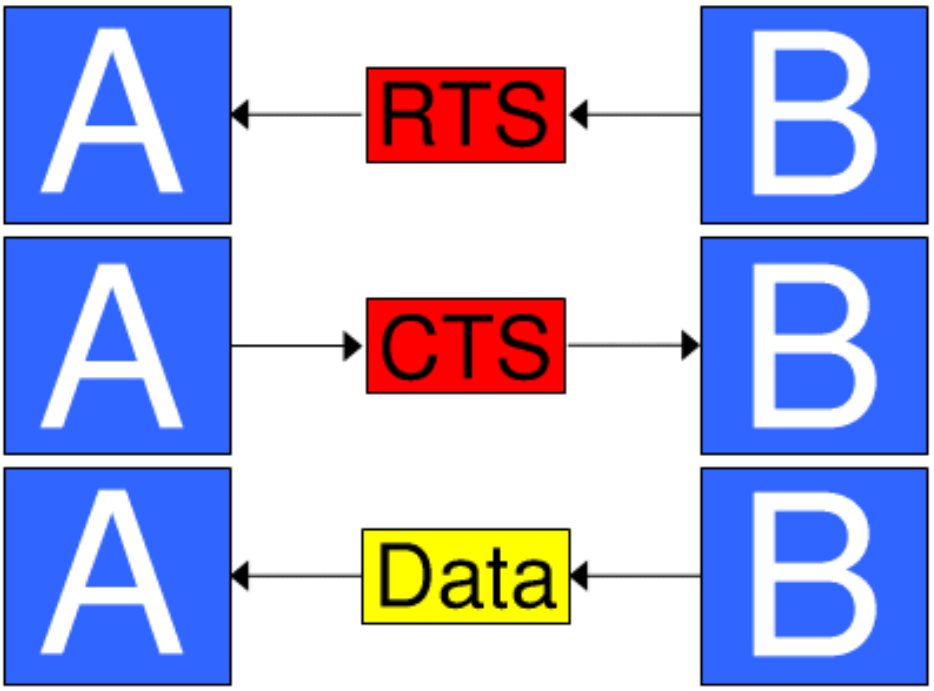


Figure 5.2: A Rendezvous Mode Transfer Between Two SCMP Nodes

the programmer must ensure that the correct receive operation executes on the destination node before the source node sends the message. The Rendezvous mode, however, sacrifices message latency for programmability by executing a two-way handshake before each message is sent. The Rendezvous message exchange protocol is illustrated in Figure 5.2. Under the Rendezvous mode, the source node will transmit a Request to Send, or RTS, message to the destination node, and wait for a Clear to Send, or CTS, message from the destination node before it transmits the data message. This two-way handshake ensures that the destination node is prepared to accept the data.

Many message-passing systems also offer a Buffering mode [4, 16, 22]. When using a Buffering mode, if a node receives a data message before any of its threads executes a receive for it, the node places the data in buffer space maintained by the messaging layer. Later, when a thread executes a receive for the message, it is copied out of the messaging layer's buffer space and into the process's buffer.

Buffering modes are typically used in conjunction with some flow control protocol. Many flow control protocols use negative acknowledgment, or NACK, messages. Under flow control protocols of this type, when a node receives a message it cannot buffer, the messaging layer at the destination node discards the message and sends a NACK to the source node. The NACK message prompts the source node to retransmit the message, possibly at a later time. Flow control protocols of this type work well on systems with enough memory to ensure that buffer shortages and retransmissions rarely occur. However, if system memory is small and shortages occur often, data retransmissions may cause too much network traffic. Handshake protocols, similar to the one used by the Rendezvous mode, offer better performance for such systems.

SCMP nodes contain an unusually small amount of memory, only 8MB, compared to most message-passing machines. Such limited memory resources guarantee that buffer shortages will be frequent. Therefore, a Buffering mode for SCMP would have to use a handshake flow control protocol. Moreover, because the messaging layer's buffer space would remain nearly full throughout most programs, CTS messages would be generated at a rate only marginally better than the rate of receive operations. Notice the Rendezvous method, already proposed for SCMP, generates CTS messages at the same rate as receive operations. Additionally, all Buffering mode implementations can cause an additional message copy operation when the message passes through the messaging layer's buffer space. For these reasons, the SCMP modifications proposed in this thesis do not include a Buffering mode of send-and-receive based message-passing.

5.2 Message Tables

As with the original active-messages message-passing system, the NIU implements the send-and-receive message-passing protocol in hardware. The implementation centers around two

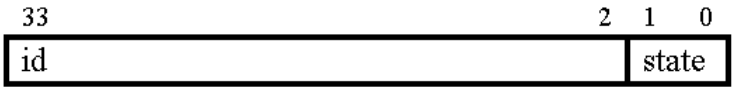


Figure 5.3: Send Table Entry

Table 5.1: Send Table Entry States

Value	State
00	Empty
01	In Use
10	In Progress
11	Complete

tables, the receive table and the send table, maintained by the NIU. The tables allow the NIU to keep records of send and receive operations, as well as flow control events, so that it may respond to message events appropriately. Additionally, SCMP threads have access to their node's send and receive tables so that they may check the status of send and receive operations. These tables are fully associative caches. This table construction allows the NIU to quickly retrieve an entry by message ID.

Figure 5.3 contains the format of entries in the send table. The send table exists only to provide records for SCMP threads to check the status of sent messages. Therefore, send entries need only a message ID field and a state field. The message ID field must be large to ensure that programmers can create as many unique messages IDs as they need with minimal effort. The specific size of thirty-two bits was chosen to support an implementation of MPI for SCMP should such an implementation ever seem appropriate. MPI identifies messages by a communicator ID and a message ID, each of which are sixteen bits long. The state field needs two bits to represent the four possible states a send entry may be in.

The four send entry states are listed in Table 5.1. The Empty state indicates that the entry

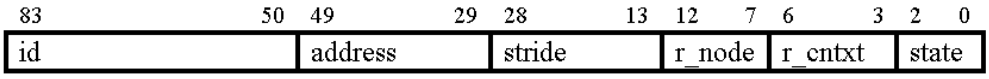


Figure 5.4: Receive Table Entry

Table 5.2: Receive Table Entry States

Value	State
000	Empty
001	In Use
010	In Progress
011	RTS Received
10X	NOT USED
110	NOT USED
111	Complete

is not in use. The NIU returns a value of Empty when a thread polls the status of a send operation not recorded in the table. The Complete state signals the polling thread that the message transfer has completed and the entry may be cleared. The In Use and In Progress states are redundant from the thread's perspective. Both mean that a thread has initiated a message transfer with the entry's ID and that transfer has not completed. However, these two states are needed by the NIU to handle errors in the control flow protocol used by the Rendezvous mode.

Figure 5.4 contains the format of entries in the receive table. The message ID field is the same as it is in the send table. The NIU uses the address and stride fields, entered by the programmer through a receive operation, when a data message arrives. The 8MB SCMP memory module contains 2^{23} addresses. However, because SCMP flits hold a thirty-two bit payload, SCMP messages are naturally word aligned. Consequently, the last two bits of a message buffer address must be zero and can be left out of the receive table entry. Therefore,

the address field requires only twenty-one bits.

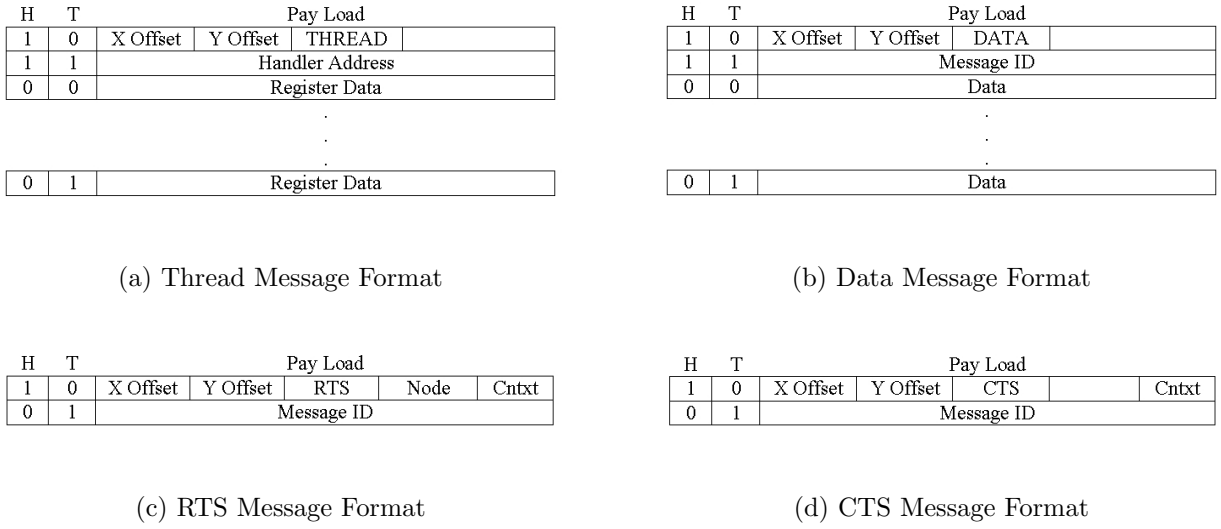
Receive entries can be in one of five states. Therefore, the state field of a receive entry must contain three bits to represent all five possible states. The five receive entry states are listed in Table 5.2. The Empty and Complete states carry the same meaning as they do for a send operation. The In Use and In Progress states are redundant from the thread's perspective. Both mean that a thread has executed a receive operation for message with the entry's ID and the message has not yet arrived. Only the NIU uses these states.

The NIU also uses the RTS Received state for internal uses only. This state indicates that the NIU has received an RTS message for the entry's message ID, but no thread has executed a corresponding receive. The r_node and r_cntxt fields only have meaning when the entry is in the RTS Received state. These fields record the source node and thread number respectively of the RTS source so that the NIU may respond with a CTS message when the correct receive operation is executed.

5.3 Message Types and Formats

To support these two modes of send-and-receive based message-passing, RTS and CTS message types are added to the SCMP message-passing protocol and the original Data message type has been modified. Notice, however, that the Thread message type remains nearly unchanged. One of SCMP's most unique and important features it its ability to quickly create threads of execution on remote nodes. Moreover, to date, there have been no major issues with the thread message protocol. Therefore, this part of the SCMP design remains unchanged.

The original SCMP Thread and Data messages use two bits, the head and tail flags, to represent only three flits types: head flits, body flits, and tail flits. By convention, the NIU



(a) Thread Message Format

(b) Data Message Format

(c) RTS Message Format

(d) CTS Message Format

Figure 5.5: New SCMP Message Formats

hardware interprets the first data flit as an address flit. The new message-passing system abandons this convention. It includes, instead, a new tag flit in Thread and Data messages. Setting both the head and tail flags of a flit equal to one marks it as a tag flit. For Thread messages, a tag flit contains the address of the thread's handler routine, thereby serving the same purpose as the address flit under the original message-passing protocol. For Data messages, however, a tag flit contains the message ID.

The initial version of the SCMP send-and-receive based message-passing protocol proposed in this thesis includes only one tag flit per Thread or Data message. Therefore, the function of the tag flit is nearly identical to the unmarked address flit of the original active-messages based protocol. However, by marking the tag flit, a relatively small design change, it may be possible to include multiple tag flits in a single Data message. This feature may be exploited by future versions of the send-and-receive based message-passing system to transmit data from and to disjoint message buffers with a single message without first packing the data into a continuous buffer.

Figure 5.5(b) displays the modified Data message format. The modified format replaces the stride field of the header flit, and the address flit, with the tag flit. Using the message ID in the tag flit, the NIU can retrieve the storage address and stride from the receive table. Figures 5.5(c) and 5.5(d) show the format of RTS and CTS messages respectively. These messages are used in the Rendezvous mode handshake protocol shown in Figure 5.2. The wormhole routing protocol used by the SCMP network requires each message to contain a head flit and a tail flit. Therefore, because RTS and CTS message are only two flits long, they do not contain a tag flit. Instead, these message formats carry the message ID to which they refer in the tail flit. Additionally, RTS messages contain the source's node ID and thread context ID. The receiving NIU uses these values to return a CTS message to the RTS source when appropriate. The CTS message includes the context ID of the thread to which it grants permission to transmit.

5.4 Assembly Level Messaging Instructions

To support the new SCMP send-and-receive message-passing library, the SCMP hardware must allow programmers to access the send and receive tables, and provide a means to pass the NIU message data. Many possible mechanisms to provide this access exist. However, because the SCMP pipeline architecture is already established, a method that requires as little modification to the current architecture as possible was chosen. Table 5.3 shows a simple modification to the original SCMP assembly level messaging instruction set, described in Chapter 3. By adding five new instructions and modifying one old instruction, this new instruction set provides all the necessary access with minimal impact on the SCMP pipeline design.

The send-and-receive based message-passing system does not require the source node to know the address or stride used by the destination node to store the contents of a data message.

Table 5.3: Modified SCMP Assembly Level Messaging Instructions

Opcode	Arguments	Description
`sendh`	d_node, type, handler	send a thread header flit
`sendh`	d_node, type, message_id	send a data header flit
`send`	data	send one data flit
`send2`	data, data	send two data flits
`sende`	data	send one data flit and end message
`send2e`	data, data	send two data flits and end message
`sendm`	l_address, {l_stride & count}	send data block from memory
`sendme`	l_address, {l_stride & count}	send data block from memory and end message
`ldss`	dest, message_id	load the state of a send
`ldsr`	dest, message_id	load the state of a receive
`str`	message_id, address, stride	store a receive operation to table
`rms`	message_id	remove a send from the table
`rmr`	message_id	remove a receive from the table

Additionally, the stride for Thread messages is always equal to one. Therefore, the `sendh` instruction in the new instruction set no longer requires a stride argument.

The new instruction set introduces five new instructions to allow threads to access the send and receive tables. The `ldss` and `ldsr` instructions allow threads to load the state of a send or receive table entry into a register. This allows threads to check for completion of send and receive operations, or to verify that a given message ID is free for use. Once a thread notices that a send or receive operation has completed, it must remove that operation's entry from the table with either the `rms` or `rmr` instruction in order to use the message ID again. The `sendh` instruction prompts the NIU to create an entry into the send table. However, threads place entries into the receive table themselves. The `str` instruction allows threads to store an address and stride pair to a receive table entry. Therefore, under the modified instruction set, the `str` instruction is a receive operation.

5.5 Modification of `sendm` and `sendme` Instructions

The new assembly level messaging instruction set contains one modification that is not directly related to the send-and-receive based message-passing protocol. Under the original SCMP messaging instruction set, the NIU retrieves the contents of the message buffers referenced by `sendm` and `sendme` instructions and places them in the transmission queue as flits before accepting further messaging instructions from the pipeline. Any messaging instructions issued by the pipeline before the `sendm` or `sendme` instruction completes will cause the thread to suspend. Often, these message buffers contain many more values than the transmission queue can hold at one time. Therefore, threads generally suspend many times as they wait for message buffers to move through the transmission queue and into the network.

To alleviate this problem, the new messaging instruction set borrows from the InfiniBand [12]

architecture which enqueues message instructions instead of message data. Under the new SCMP messaging instruction set, the NIU enqueues two temporary pseudo-flits when it receives a `sendm` or `sendme` instruction. The first pseudo-flit contains the message buffer address, and the second contains the message stride and the number of values to send. Once these pseudo-flits reach the head of the transmission queue, the NIU begins to move data from memory directly into the network using the parameters they contain. The NIU does not pull any more flits from the transmission queue until the memory transfer has completed. This new `sendm` and `sendme` behavior allows threads to execute a number of send operations without suspending.

5.6 Rendezvous Mode Operation

Consider the Rendezvous mode data transfer from node B to node A depicted in Figure 5.2. A thread on node B, say thread one, initiates this transfer by performing a send operation. To perform the send operation, thread one submits a message to the NIU through a sequence of assembly instructions from the `send` family. This sequence must begin with a `sendh` instruction. The `sendh` instruction prompts the NIU on node B to check the entry in its send table corresponding to the message ID provided in the instruction. If the table does not contain an entry for that message ID, the NIU enqueues a head and tag flit in the transmission queue and places an entry in the table. Otherwise the send operation is rejected, and thread one suspends. Once the head and tag flit pair reach the head of the queue, the NIU on node B will inspect their contents and transmit an RTS message to node A.

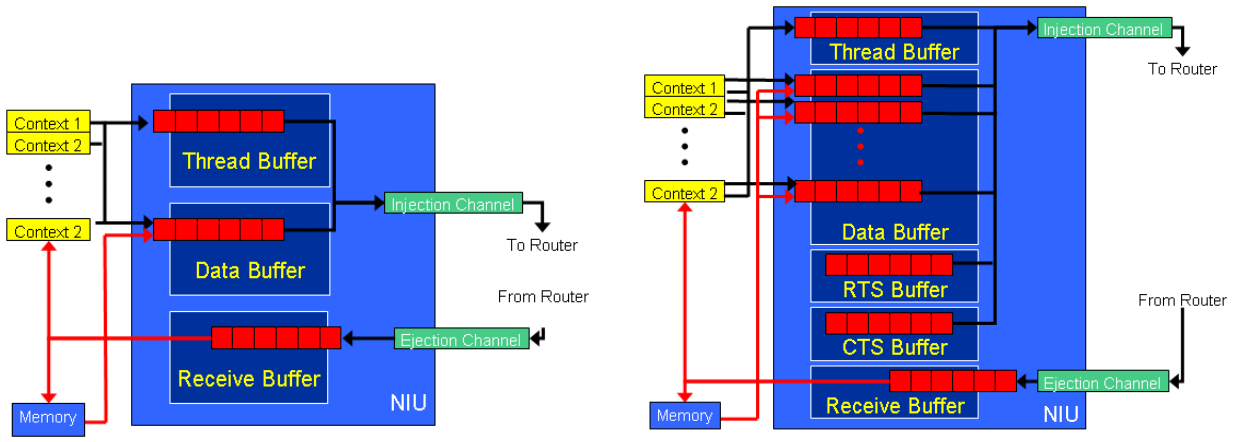
When the NIU on node A receives the RTS message, its reaction depends upon the state of the receive table entry corresponding to the message ID contained in the message. If the table does not contain the appropriate entry, then no thread on node A has a pending receive operation for that message ID. Therefore, the NIU will place a new entry in the table to

store the source node ID and thread context ID from the RTS message. This new entry will begin in the RTS Received state. This will enable the NIU to generate a CTS message to the appropriate sender once a thread on node A executes a receive operation for this message ID. If an entry exists in the In Use state, however, the NIU will immediately transmit a CTS message to the sender indicated in the RTS message and move the entry's state to In Progress.

A thread on node A, say thread two, accepts this transfer by performing a receive operation. To perform the receive operation, thread two executes an `str` instruction. Upon receiving the `str` instruction, the NIU on node A will check the entry in its receive table corresponding to the message ID provided in the instruction. If it does not find an entry, or if the entry's state equals RTS Received, the NIU stores the address and stride provided into the table. Otherwise, the operation is rejected and thread two suspends. If the entry's state equals RTS Received the NIU additionally transmits a CTS message to the source recorded in the entry.

5.6.1 RTS Messages and Deadlock Avoidance

When an entry's state equals In Progress or RTS Received, the NIU has established the source for the current transmission of a message corresponding to that entry. Therefore, any additional RTS messages that arrive with the same message ID must be blocked until the current transmission completes. By blocking messages in the network the NIU creates the possibility of deadlock. As stated previously, however, the dimension order routing used by SCMP will remain deadlock free as long as messages continue to drain from the network. When RTS messages block, they form a queue in the network. Each RTS message in this queue may leave the network once the transfer associated with the RTS message ahead of it completes. Therefore, for blocked RTS messages to drain, they must not prevent CTS and Data messages to pass through the network. To ensure RTS messages do not block the



(a) Original NIU Architecture

(b) Modified NIU Architecture

Figure 5.6: Block Diagrams of the Original and Modified NIU

progress of other message types, the modified SCMP architecture gives each message type its own virtual channel partition.

5.6.2 Rendezvous Handshakes on Multi-Threaded Nodes

After an NIU transmits an RTS message it cannot inject any flit from the data transmission queue until it receives a CTS message from the destination. Revisiting the example transfer between thread one on node B and thread two on node A, this will cause a significant performance penalty if thread one is not the only active thread on node B. If the NIU contains only one transmission queue for all data messages, other threads on node B will be unable to transmit data messages until thread one's message completes. Depending upon the activity of thread two on node A, this may take a long time. Therefore, the modified SCMP architecture includes a separate data transmission queue for each thread context. Figure 5.6 contains block diagrams for the original NIU architecture and the modified NIU architecture.

These data transmission queues are multiplexed across the injection channel in first come first serve order. During every clock cycle, the NIU checks the state of each data transmission queue. Each queue may either be ready, waiting, or empty. A queue is in the empty state until a head flit arrives at the head of the queue. After the NIU transmits an RTS message for a queue, the queue moves to the waiting state. Once the NIU receives a CTS message for a waiting queue, the NIU moves the queue to the ready state and places the queue on a waiting list for the injection channel. Once a queue gains control of the injection channel, it maintains control until it transmits one message. After this transmission, the queue returns to the empty state.

5.7 Ready Mode Operation

Because it does not include a flow control protocol, Ready mode operation is much simpler. Consider the Ready mode data transfer from node B to node A depicted in Figure 5.1. A thread, say thread one, initiates the transfer in the same way as a Rendezvous transfer. The NIU similarly checks the send table and suspends thread one if necessary. Once the head flit reaches the head of thread one's transmission queue, however, the NIU does not transmit an RTS message to node A. Instead, thread one's queue moves immediately to the ready state. Once thread one's queue gains control of the injection channel, the NIU transmits the Data message without any flow control handshake.

When the NIU on node A receives the Data message, its reaction depends upon the state of the receive table entry corresponding to the message ID contained in the message. If the table does not contain the appropriate entry, then no thread on node A has a pending receive operation for that message ID. Under Ready mode, this scenario can only result from programmer error. Therefore, node A's NIU will discard the message. Additionally, the NIU may trigger an exception to notify the programmer of the error. If the table contains an

entry in the In Use state, however, the NIU will begin to store the message according to the entry's address and stride fields and move the entry to the In Progress state. Once storage completes, the NIU moves the entry to the Complete state. If the entry's state equals In Progress when the Data message arrives, the Data message will block until the previous transmission completes.

Chapter 6

Stressmark Testing

To verify that the modifications made to the SCMP message-passing system and programming model do not result in lower processor performance, the modified architecture was tested under a handful of stressmarks. These stressmarks include the neighborhood, matrix, and transitive closure stressmarks from the Data-Intensive Systems Stressmark Suite [2], as well as the popular LU factorization stressmark [5]. To keep the comparison fair, the algorithms and parallelization techniques used in these benchmarks remain nearly the same. Only the programming model and message-passing methods are altered. As the results presented in this chapter show, the modified SCMP architecture performs as well as, and often better than, the original design. Stressmarks written under SCMP's send-and-receive programming model require 0.9% to 51.02% less clock cycles to complete than those written under the original active-messages programming model.

6.1 The SCMP Simulator

To test the performance of the SCMP architecture as it evolves, without fabricating examples of each design iteration, the SCMP research group developed a functional simulator for SCMP written in C [18]. This simulator's modular design allows developers to create a new simulator version to model a modified SCMP architecture by rewriting the pertinent modules. To model the architectural modifications described in Chapter 5, the modules that model the SCMP NIU, router, and instruction dispatch components were rewritten.

6.2 DIS Stressmarks

As part of the DARPA Information Technology Office's Data-Intensive Systems research program, the Atlantic Aerospace Electronics Corporation produced the DIS Stressmark Suite in 1999 [2]. This stressmark suite contains algorithms designed to frequently access data sets in non-contiguous patterns. SCMP seems particularly suited to these types of applications because it integrates memory hardware into every node without any cache hierarchy between the pipeline and memory interface. Additionally, the SCMP architecture has the ability to distribute the data among many nodes, and efficiently issue threads on any node to manipulate that data.

Many of the DIS stressmarks have complex message exchange patterns. Guaranteeing that all receive operations execute before their corresponding send operations is impractical for these stressmarks. Therefore, the DIS stressmarks implemented under the new send-and-receive based programming model use the Rendezvous message-passing mode.

6.2.1 Neighborhood Stressmark

Description

The neighborhood stressmark represents an operation often used by image processing applications to gather texture information about an image. The stressmark takes a square grayscale image as input, and measures its texture in terms of two Gray-Level Co-occurrence Matrix statistical descriptors. Namely, these descriptors are the image's GLCM entropy and GLCM energy.

The neighborhood stressmark estimates the image's GLCM entropy and energy by building two histograms. One histogram represents the sum of pairs of pixel values, and the other represents the differences between pairs of pixel values. Pairs of pixels are defined at four different angles and two different distances. For each pair class, defined by the angle and distance, the stressmark builds a sum-histogram and a difference-histogram. The stressmark then uses these histograms to estimate the GLCM entropy and energy for each pixel pair class.

To parallelize this algorithm for SCMP, rows of the image are distributed evenly among the nodes. Each node has its own copy of the sum-histogram and difference-histogram. For each row that a node owns, that node will iterate through the pixels of the row and determine each pixel's pair according to the current angle and distance parameters. If the node does not own the row in which the other pixel lies, it will request the row from its owner by initiating a thread on that node. Once it has both pixel values, the node calculates their sum and difference and increments the appropriate slots in the sum- and difference-histograms.

Once all of the nodes complete the calculations for their rows, the histograms on each node must be combined. To reduce the amount of data that must be transmitted during this combination, each node takes responsibility for only a section of each of the combined

histograms. A node will only receive these portions of the histograms located on other nodes. Each node then calculates the GLCM entropy and energy for its portion of the histograms. The global GLCM entropy and energy values are calculated through a global reduction.

When programmed for the original SCMP programming model, nodes running the neighborhood stressmark request rows and histogram portions from other nodes by creating threads on those nodes. When programmed under the send-and-receive based programming model, histogram portions are no longer distributed via requests. Every node can easily calculate the owner of each histogram portion. Therefore each node can send the appropriate portions of its histograms to every other node without first receiving a request. During the histogram distribution phase, each node spawns a local thread to send the correct histogram portions to every other node. Meanwhile, the main thread executes receive operations for the histogram portions owned by the local node.

Requests are still used to receive image rows owned by remote nodes, however. As a node iterates through its pixels, it knows what node owns that pixel's pair. Therefore, letting the node request the remote row creates a more straightforward program. The request threads and thread messages used to request rows do not create a significant performance impact because the network traffic generated by row exchanges is very small in comparison to that generated by histogram distribution.

Data Transmission Queue Depth

As mentioned in Chapter 5, the modified SCMP architecture includes a data transmission queue for each thread context. To determine the appropriate depth for these queues, the neighborhood benchmark was run with the simulator configured to use a variety of transmission queue depths. Figure 6.1 shows the speedup achieved by the neighborhood stressmark for data transmission queue depths ranging from two to one-hundred twenty-eight flits. The

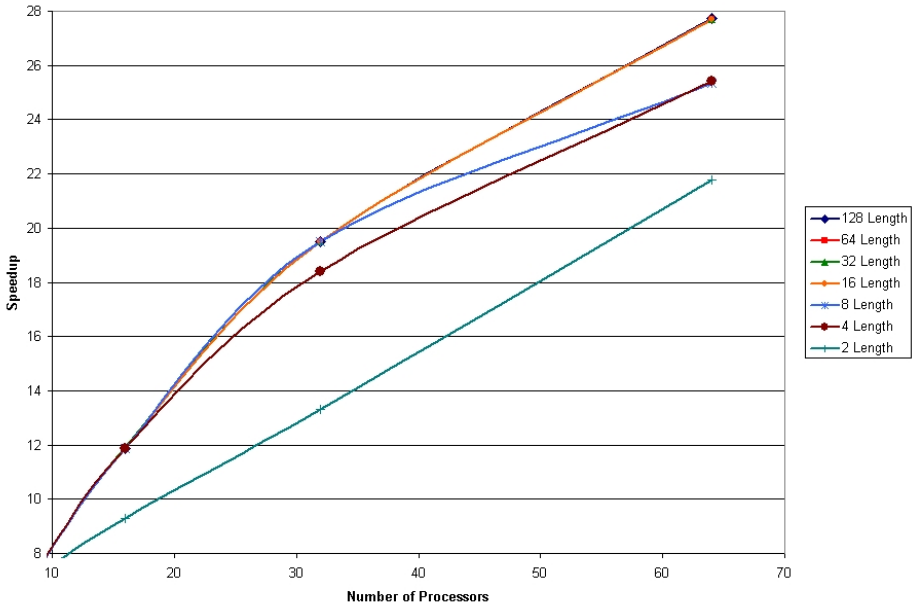


Figure 6.1: Speedup Curves for Neighborhood Benchmark With Varied Transmission Queue Lengths

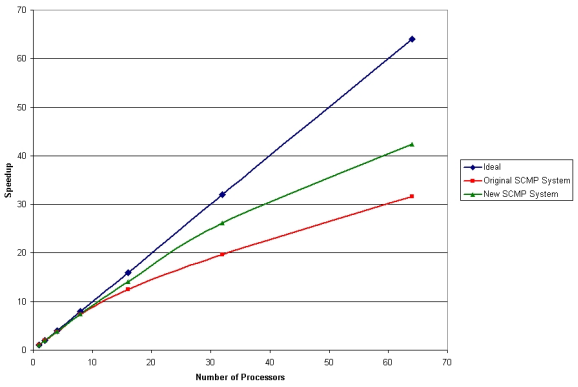
number of bits used to represent each pixel in the neighborhood stressmark's input image may range from seven to fifteen bits. For the queue depth experiment, the input file contained eleven-bit wide pixels.

As shown in Figure 6.1, increasing the queue depth beyond sixteen flits does not create increased performance. The speedup curves produced by data transmission queue depths of sixteen flits to one-hundred twenty-eight flits are identical. However, queue depths smaller than sixteen flits do not produce equally desirable speedup curves. As the number of SCMP nodes increases, each node must communicate with more remote nodes; thereby enqueuing more messaging instructions. Figure 6.1 shows that data transmission queues smaller than sixteen flits become a bottleneck as the number of SCMP nodes increases.

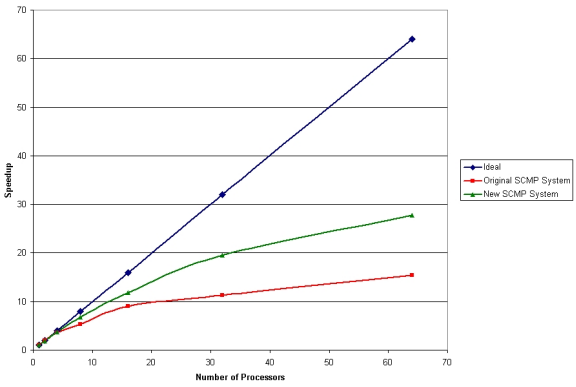
Performance

Figure 6.2 shows the performance results of the neighborhood stressmark when run on input files with seven, eleven, and fifteen bit wide pixels. Each graph in Figure 6.2 contains an ideal speedup curve, the speedup curve of the original SCMP architecture, and the speedup curve of the modified SCMP architecture. With an input file containing seven-bit wide pixels, the send-and-receive version of the neighborhood stressmark produces a peak runtime improvement over the original active-messages version of 25.49% when run on sixty-four SCMP nodes. Using an input file containing eleven-bit wide pixels, the send-and-receive version reaches a peak runtime improvement of 44.4% when run on sixty-four nodes. The number of communication partners has a greater performance impact when an input file containing fifteen-bit wide pixels is used. With an input file of this size, the send-and-receive version of the neighborhood stressmark produces a peak performance improvement of 51.02% when run on sixteen nodes.

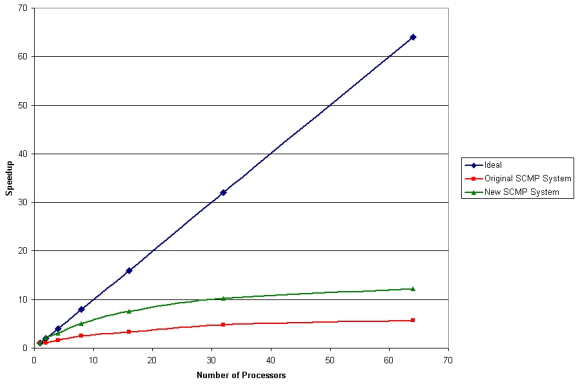
When the width of the input pixels increases by one, the size of the sum- and difference-histograms increases by a factor of two. For example, the sum- and difference-histograms needed to manage an input file containing fifteen-bit wide pixels contain two-hundred fifty-six times the number of bins present in the sum- and difference-histograms needed for an input file with seven-bit wide pixels. Consequently, more data must be exchanged among nodes. The performance impact this increased message traffic creates is clearly visible in the figure. Still, the modified SCMP architecture performs better than the original on all input files.



(a) Seven-Bit Pixels



(b) Eleven-Bit Pixels



(c) Fifteen-Bit Pixels

Figure 6.2: Speedup Curves for Neighborhood Benchmark Under Original and New SCMP Message-Passing Systems

6.2.2 Conjugate Gradient

Description

The matrix stressmark uses an iterative conjugate gradient method to solve the general linear system expressed in Equation 6.1, where A is a sparse matrix.

$$A \cdot x = b \tag{6.1}$$

To perform the conjugate gradient, the stressmark defines two vectors, r_k and p_k, where k represents the iteration number. With these two vectors, the stressmark forms an improving sequence of estimates using Equation 6.2. The stressmark input file specifies an acceptable error tolerance and a maximum number of iterations to perform. Equation 6.3 expresses the error of estimate x_k. The stressmark continues to produce estimates until the error of x_k falls within the error tolerance or k reaches the maximum iteration number.

$$x_{k+1} = x_k + \alpha_k p_k \tag{6.2}$$

$$error = \frac{|A \cdot x - b|}{|b|} \tag{6.3}$$

The stressmark initializes r_k and p_k with vector b, that is $r_1 = p_1 = b$, and initializes all values of x to 0. Equations 6.4 through 6.7 show the calculations performed during each iteration to update r and p.

$$\alpha_k = \frac{r_k^T \cdot r_k}{p_k^T \cdot (A \cdot p_k)} \tag{6.4}$$

$$r_{k+1} = r_k - \alpha_k A \cdot p_k \tag{6.5}$$

$$\beta_k = \frac{r_{k+1}^T \cdot r_{k+1}}{r_k^T \cdot r_k} \tag{6.6}$$

$$p_{k+1} = r_{k+1} - \beta_k p_k \tag{6.7}$$

To parallelize this algorithm for SCMP, rows of matrix A and portions of vectors b and r_k are distributed cyclically among the nodes. Every node keeps complete p_k and x_k vectors but only manipulates its portion of p_k. Node 0 governs the computations performed on every other node. During each iteration, node 0 creates a thread on every node to execute Equations 6.2 through 6.7, or parts of these equations. Results of these calculations are returned to node 0 through global reductions, and node 0 redistributes the values as needed in subsequent thread messages.

To calculate α_k node 0 asks every node to calculate $A \cdot p_k$ on its rows of A. To do so, however, every node must have a complete and up-to-date copy of p_k. Therefore, at the end of each iteration, every node must share its portion of p_k with all other nodes. To reduce the number of messages required for this redistribution, nodes first distribute their portions of p_k with nodes in the same column. After this step, each node has updated values for all portions of p_k owned by any node in the same column. By sharing all of these values with nodes in the same row, every node will receive all remaining portions of p_k.

Under the original SCMP programming model, nodes request portions of p_k by starting threads on nodes in the same column during the column distribution step, and on nodes in the same row in the row distribution step. Under the new programming model, however, the matrix stress mark does not need to use any request threads. Instead, each node executes

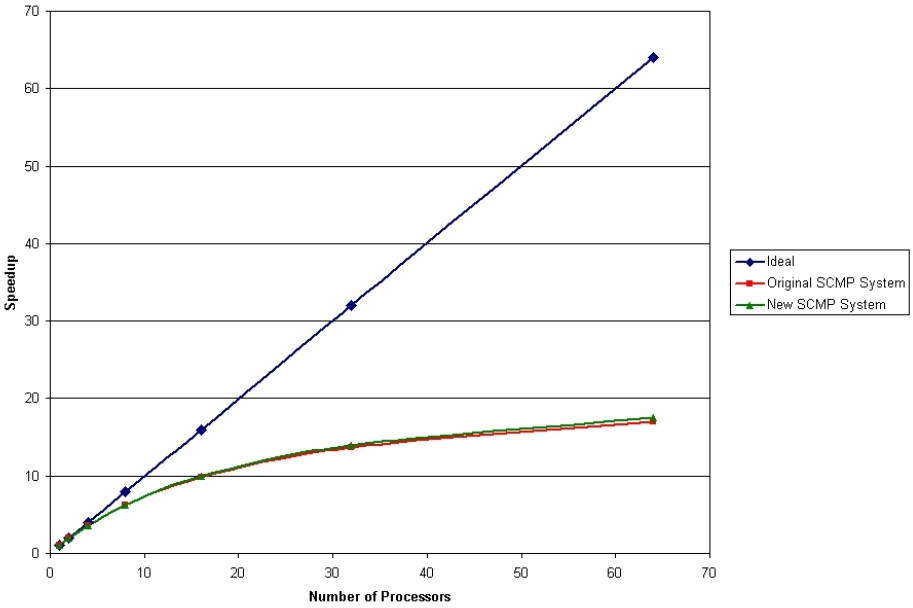


Figure 6.3: Speedup Curves for Matrix Benchmark Under Original and New SCMP Message-Passing Systems

a non-blocking receive operation for messages from nodes in the same column during the column distribution step, or nodes in the same row during the row distribution step. Each node then sends its column or row neighbors, depending on the distribution step, all updated portions of p_k it currently holds.

Performance

Figure 6.3 shows the performance results of the matrix stressmark when run on a sparse matrix with ten-thousand rows. Figure 6.3 contains an ideal speedup curve, the speedup curve of the original SCMP architecture, and the speedup curve of the modified SCMP architecture. Both the original and modified SCMP architectures produce speedup curves well below the ideal speedup for this stressmark due frequent distributions of p_k. The k=large volume of data that the stressmarks move through the network impacts performance much

more than the overhead of the messaging system. Therefore, the improvments in overhead offered by the send-and-receive message-passing system did not significantly improve the overall performance. The send-and-receive version of the matrix stressmark produces a peak runtime improvement over the original active-messages version of 3.21% when run on sixty-four SCMP nodes.

6.2.3 Transitive Closure

Description

The transitive closure stressmark uses the Floyd-Warshall algorithm [9] to find the shortest distance between all pair of vertices in a directed graph of size n. The stressmark constructs a representation of an arbitrary directed graph at runtime by filling an n by n adjacency matrix with random values. The algorithm requires n iterations to complete. During each iteration k, the stressmark replaces each adjacency matrix element $D[i][j]$ with $\min(D[i][j], D[i][k] + D[k][j])$.

To parallelize this algorithm for an X by Y grid of SCMP nodes, the adjacency matrix is divided into $\frac{n}{X}$ by $\frac{n}{Y}$ sub-blocks. Each node owns a sub-block. For each iteration, k, each node needs a portion of row k and column k of the adjacency matrix. In particular, if a node's sub-block contains portions of columns r through s of the adjacency matrix, then it will need elements r through s of row k. It will need a similarly defined portion of column k.

Under the original SCMP programming model, each node requests the portion of row k and column k it needs by starting a thread on the nodes that own those portions. Notice, however, that nodes in the same column of the SCMP grid own identical columns of the adjacency matrix and all nodes in the same row of the SCMP grid own identical rows of the adjacency matrix. Therefore, under the new programming model the nodes that own

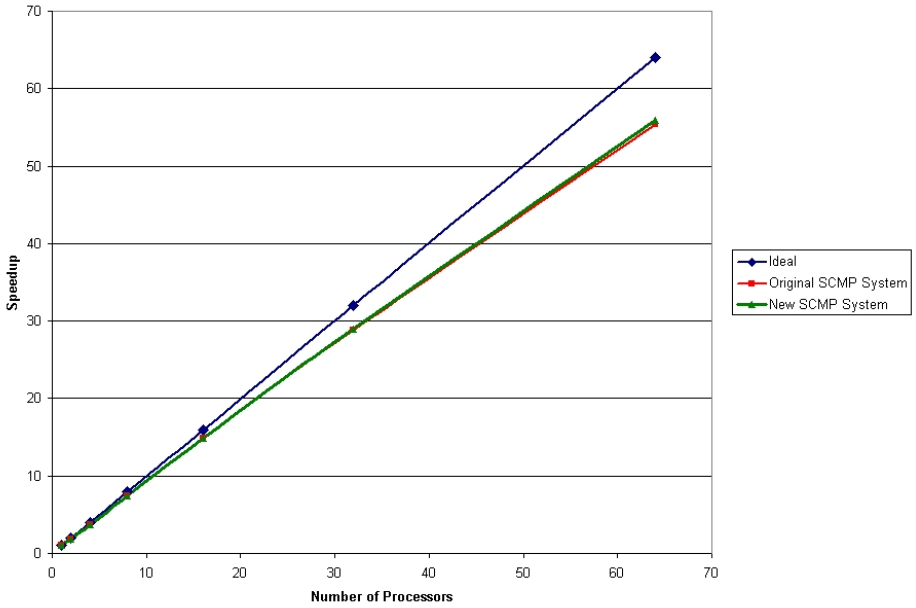


Figure 6.4: Speedup Curves for Transitive Closure Benchmark Under Original and New SCMP Message-Passing Systems

portions of row k send the portion to every node in the same grid column. Similarly, the owners of portions of column k send the portion to every node in the same grid row. Every node executes a receive operation for a message containing a row k portion and a message containing a column k portion.

Performance

Figure 6.4 shows the performance results of the transitive closure stressmark when run on a directed-graph containing five-hundred vertices. Figure 6.4 contains an ideal speedup curve, the speedup curve of the original SCMP architecture, and the speedup curve of the modified SCMP architecture. The transitive closure stressmark exhibits a very low communication to computation ratio. Therefore, this stressmark allowed little room for the send-and-receive message-passing system to improve the overall performance. Both original and modified

SCMP architectures show impressive speedup curves for this stressmark. The send-and-receive version of the transitive closure stressmark produces a peak runtime improvement over the original active-messages version of 0.9% when run on sixty-four SCMP nodes.

6.3 LU Factorization Stressmark

6.3.1 Description

The LU factorization stressmark factors a dense matrix A into a lower triangular matrix L and an upper triangular matrix U such that A = LU. This stressmark is used as a kernel in applications that solve linear systems of equations. The stressmark divides the dense matrix A into a number of square blocks, and iterates through the blocks that lie on the diagonal. The following pseudo-code describes the stressmark's operation:

```
LU FACTORIZATION(A)
 1  N ← DIM(A)
 2  for k ← 0 to N −1 /*loop over all diag blocks*/
 3        do
 4           factorize block A_{k,k}
 5           for j ← k +1 to N −1 /*divide perimeter blocks by diag block */
 6                 do
 7                    A_{k,j} ← A_{k,j} * (A_{k,k})^{-1}
 8                    for i ← k +1 to N −1 /* modify inner matrix blocks by perimeter blocks */
 9                          do
10                             A_{i,j} ← A_{i,j} − A_{i,k} * (A_{k,k})^T
```

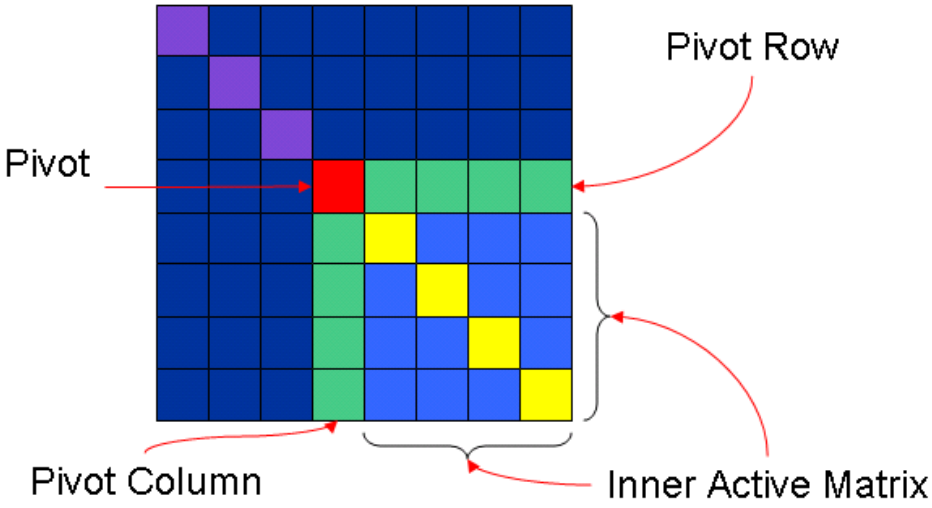


Figure 6.5: LU Factorization Block Decomposition

To parallelize the algorithm for SCMP, the blocks of A are distributed among the SCMP nodes in a cyclic fashion. Each iteration begins with the factorization of the current diagonal block. After the diagonal block is factored, the owners of the perimeter blocks divide the perimeter blocks by the diagonal block. Perimeter blocks are blocks of A to the left of the diagonal block in the same row of A, and below the diagonal block in the same column of A. After all of the perimeter blocks have been divided, each block in the inner-active matrix must be modified by the perimeter block directly above it and directly to its left. The inner-active matrix is the portion of A below the perimeter row, and to the right of the perimeter column.

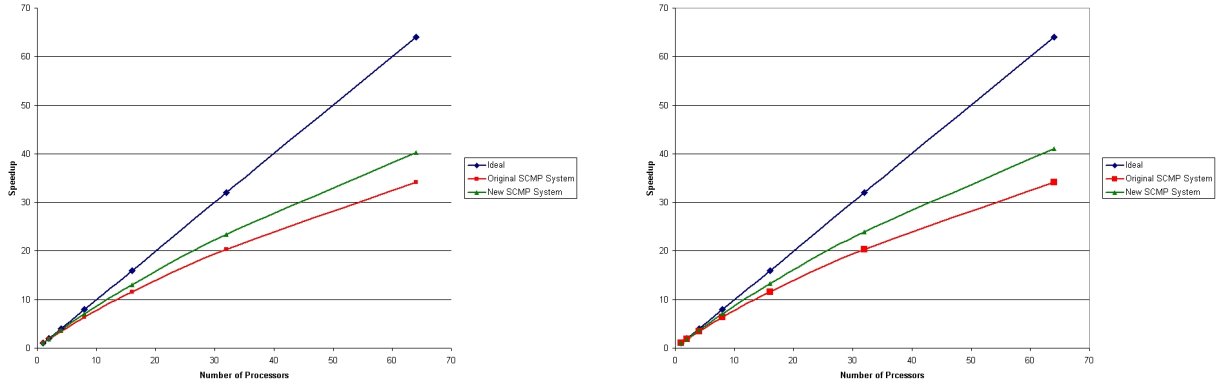
Under the original SCMP programming model, nodes request the diagonal block and perimeter blocks by creating threads on the nodes that own these blocks. To prevent the owners the diagonal and perimeter blocks from becoming overwhelmed with threads, barriers are used to separate diagonal block request periods from perimeter block request periods, as well as to separate each iteration from the next. Under the new SCMP programming model, nodes post non-blocking receives for all the blocks that they will need at the beginning of each iteration. Two version of the LU factorization stressmark were written under the send-

and-receive programming model. One of these uses the Ready message-passing modes and the other uses the Rendezvous message-passing mode. The Ready mode version of the LU factorization stressmark includes a barrier to guarantee that every node finishes posting all of its receives for the current iteration before any node begins to execute send operations.

Under the cyclic block distribution scheme used in the LU stressmark, the owner of the diagonal block is in the same row of the SCMP grid and the owners of the perimeter row blocks, and the same column of the SCMP grid as the owners of the perimeter column blocks. Similarly, the perimeter blocks needed to modify an inner-active matrix block $A_{i,j}$ are owned by nodes in the same SCMP grid row and column as the owner of block $A_{i,j}$. Therefore, after it has factored the diagonal block, the diagonal block owner sends the block to all nodes in its SCMP grid row and column. Once the owners of the perimeter row blocks have received the diagonal block, and divided the perimeter block by it, they send their perimeter block to all nodes in the same SCMP column. Similarly, once the owners of the perimeter column blocks have received the diagonal block, and divided their perimeter block by it, they send the perimeter block to all nodes in the same SCMP row.

6.3.2 Performance

Figure 6.6 shows the performance results of the LU factorization stressmark when run on a five-hundred and twelve by five-hundred and twelve element matrix, with sixteen by sixteen element blocks. Each graph in Figure 6.6 contains an ideal speedup curve, the speedup curve of the original SCMP architecture, and the speedup curve of the modified SCMP architecture. The performance of the LU stressmark suffers as the number of nodes increases when run on the original SCMP architecture because the owners of the diagonal and perimeter blocks become inundated with threads requesting data. However, both send-and-receive versions of the stressmark demonstrate significantly improved performance because they do not use threads to request data. Using Ready mode message-passing the send-and-receive version



(a) Ready Mode

(b) Rendezvous Mode

Figure 6.6: Speedup Curves for LU Factorization Benchmark Under Original and New SCMP Message-Passing Systems

of the stressmark produces a peak runtime improvement over the original active-messages version of 15.06%, using Rendezvous mode message-passing it produces a peak runtime improvement of 16.67%. Under both message-passing modes, the send-and-receive stressmark version produces the peak performance improvement when run on sixty-four nodes.

Chapter 7

Conclusions

7.1 Summary of Findings

This thesis has proposed a new programming model for the SCMP parallel computer, and several architectural modifications to support it. Weaknesses of the original SCMP programming model were presented as motivation for the new model. The original programming model is built on top of an Active-Messages messaging layer. Data messages in this message-passing system contain the address to which they will be stored, and they are accepted without notifying any of the threads at the destination. As a result, the original programming model requires the programmer to use global message-buffer pointers, thread messages to request data transmission, and synchronization messages to notify threads on the destination node that data transmission has completed. These methods create several problems, including an over-abundance of thread messages that request data, and race conditions between data messages and their corresponding synchronization messages.

Because it is built on top of a send-and-receive messaging layer, the new SCMP programming model alleviates these problems. The send-and-receive based message-passing system allows

threads to control the reception of data by executing receive operations. Through these receive operations, threads may provide the messaging layer with all of the information needed to accept the corresponding data message. This local information exchange eliminates the need to inform the data's source node of the storage information through a thread message. Additionally, receive operations create a local record that threads may poll to check for message arrival. Therefore, the new programming model eliminates the need for many synchronization messages.

To measure the performance impact of the new programming model and messaging layer, a functional simulator was developed for the modified architecture and used to simulate a handful of stressmarks written with the new programming model. These stressmarks included the neighborhood, matrix, and transitive closure stressmarks from the DIS stressmark suite, and the popular LU factorization stressmark. The results of these simulations were compared to the performance of the same stressmarks written for the original SCMP programming model and executed by a simulator for the original SCMP architecture. The comparisons show that the new SCMP programming model and architecture perform as well as, and often better than, the original design. The new SCMP programming model and architecture offer runtime improvements ranging 0.9% to 51.02%.

7.2 Summary of Author's Work

This thesis required creating the new programming model for SCMP and the send-and-receive based message-passing system upon which it is built. This includes designing the new programming model and creating library routines that implement it, designing the architectural modifications described in Chapter 5 and simulating these changes through modifications to the SCMP simulator, and, finally, modifying the SCMP assembler to represent the changes to the SCMP assembly level messaging instruction set. Additional work for

this thesis includes rewriting the stressmarks discussed in Chapter 6 under the new SCMP programming model and comparing their performance to that of versions written under the original SCMP programming model.

7.3 Future Work

Because SCMP flits carry thirty-two bit payloads, messages sent by either the active-messages or by the send-and-receive based message-passing systems for SCMP must be word-aligned. This requirement burdens programmers who wish to send a message with data elements larger than thirty-two bits. For example, because floating-point numbers contain sixty-four bits on an SCMP computer, a programmer who wishes to send an array of stridden floating-point values across the network must send two messages. One message contains the high word of each floating-point value while the other contains the low word of each floating-point value.

Multi-part messages are one possible solution to this problem. Multi-part messages allow the programmer to send data from and to disjoint message buffers in a single message. The tag flit included in the send-and-receive based message-passing system for SCMP opens the door for multi-part messages by providing a means for identifying sub-parts of a message. Under a multi-part messaging system, the two messages used to send an array of floating-point number in the previous example would become two sub-parts of a single message. Each sub-part must begin with a tag flit so that the NIU may retrieve the storage stride and address from the receive table for each. Therefore, the message still requires a message ID for each sub-part. Unless a system evolves to either eliminate the need for multiple message IDs for multi-part messages or produce IDs for sub-parts without the programmers input, the burden of managing multiple message IDs for multi-part messages will fall on the programmer.

Messages containing data elements smaller than thirty-two bits present more difficult challenges. Consider the efforts required of a programmer who wishes to send an array of stridden one byte values across the network. The programmer cannot safely send the data using a data message. Because SCMP messages are word aligned, for each byte that the NIU sends it will also send three other bytes adjacent in memory to the intended byte. These adjacent bytes will be written into the receiving node's memory. Therefore, the programmer will unintentionally alter some bytes in the receiver's memory if he or she uses a data message. Therefore, the programmer must send the data as arguments to a remote thread that knows how to extract and store the correct bytes,or pack the data and have some remote thread unpack it at a later time. To solve this problem, the SCMP message-passing system needs some way to indicate what parts of a flit's payload are valid, or hardware support for packing and unpacking elements smaller than thirty-two bits.

While the tests discussed in Chapter 6 are sufficient to give an indication of the performance of the modified SCMP programming model and architecture, they are by no means exhaustive. Full applications and more stressmarks should be written under the new programming model and tested by the new SCMP simulator. Further testing will form a more concrete perception of the new architecture's performance. Additionally, these tests may discover network communication bottlenecks in the new message-passing system that are currently unknown.

Bibliography

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000.

[2] Atlantic Aerospace Electronics Corporation. *Data-Intensive Systems Benchmark Suite Analysis and Specification*, 1.0 edition, June 1999.

[3] James M. Baker Jr., Sidney Bennett, Mark Bucciero, Brian Gold, and Rajneesh Mahajan. SCMP: A single-chip message-passing parallel computer. In *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1485–1491, 2002.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[5] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

[6] William J. Dally, J.A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, April 1992.

[7] William J. Dally and S. Lacy. VLSI architecture: past, present, and future. In *Advanced Research in VLSI Conference*, pages 232–241, 1999.

[8] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: an Engineering Approach*. IEEE Computer Society, 1997.

[9] R. W. Floyd. Algorithm 97 (shortest path). *Communications of the ACM*, 5(6):345, 1962.

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

[11] 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.

[12] InfiniBand Trade Association. *InfiniBand Architecture Specification*, 1.0 edition, October 2000.

[13] Christoffer Lutz, Steve Rabin, Charles L. Seitz, and Donald Speck. Design of the mosaic element. *Conference on Advanced research in VLSI*, pages 1–10, 1984.

[14] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3), 1994. ftp://www.netlib.org/mpi/mpi-report.ps.

[15] J. Miguel, A. Arruabarrena, R. Beivide, and J. A. Gregorio. Assessing the performance of the new IBM SP2 communication subsystem. *IEEE Parallel & Distributed Technology*, 4:12–22, 1996.

[16] Myricom Inc. *GM Reference Manual*, 2.0.6 edition, September 2003.

[17] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing*, pages 1–22, 1995.

[18] Priyadarshini Ramachandran, Charles W. Lewis Jr., and James M. Baker Jr. A power and performance simulator for a single-chip message-passing parallel architecture. In *The 2004 International Conference on Modeling, Simulation and Visualization Methods*, 2004.

[19] R. Seifert. *Gigabit Ethernet: Technology and Applications for High Speed LANs.* Addison-Wesley, 1998.

[20] Semiconductor Industry Association. International technology roadmap for semiconductors, 2003.

[21] Piyush Shivam, Pete Wyckoff, and Dhabaleswar Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *Supercomputing 2001*, November 2001.

[22] M. Snir, P. Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the IBM SP2. *IBM Systems Journal*, 34(2):205–221, 1995.

[23] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[24] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[25] D. Wills, H. Cat, J. Cruz-Rivera, W. Lacy, James M. Baker Jr., J. Eble, A. Lopez-Lagunas, and M. Hopper. High-throughput, low-memory applications on the pica architecture. *IEEE Transactions on Parallel and Distributed Systems*, 8:1055 – 1067, October 1997.