High Performance Applications for the Single-Chip Message-Passing Parallel Computer

William Wesley Dickenson

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

James M. Baker Jr., Chair

Mark T. Jones

Michael S. Hsiao

April 22, 2004

Blacksburg, Virginia

Keywords: Parallel Architectures, Parallel Applications, Message-Passing Systems, Single-Chip Systems, Thread Level Parallelism, Chip Multiprocessors

Copyright 2004, William Wesley Dickenson

High Performance Applications for the Single-Chip Message-Passing Parallel Computer

William Wesley Dickenson

James M. Baker, Ph.D., Committee Chair
Department of Electrical and Computer Engineering

Abstract

Computer architects continue to push the limits of modern microprocessors. By using techniques such as out-of-order execution, branch prediction, and dynamic scheduling, designers have found ways to speed execution. However, growing architectural complexity has led to unsustained development and testing times. Shrinking feature sizes are causing increased wire resistances and signal propagation, thereby limiting a design's scalability. Indeed, the method of exploiting instruction-level parallelism (ILP) within applications is reaching a point of diminishing returns.

One approach to the aforementioned challenges is the Single-Chip Message-Passing (SCMP) Parallel Computer, developed at Virginia Tech. SCMP is a unique, tiled architecture aimed at thread-level parallelism (TLP). Identical cores are replicated across the chip, and global wire traces have been eliminated. The nodes are connected via a 2-D grid network and each contains a local memory bank.

This thesis presents the design and analysis of three high-performance applications for SCMP. The results show that the architecture proves itself as a formidable opponent to several current systems.

Dedication

To my wife, parents, and brother, without your love and support none of this would have been possible

Acknowledgments

I cannot thank my advisor, Dr. James Baker, enough for all his help and advice. My research, and this thesis, would never have been accomplished without his patience, direction, and guidance.

I would also like to thank Dr. Mark Jones and Dr. Michael Hsiao for serving on my defense committee as well as providing insight on the direction of my research.

Many thanks go to Vinit Joshi and Jeff Poole for their help with MPI. In addition, I need to thank Charles (Will) Lewis, Priyadarshini Ramachandran, and the rest of the SCMP team. I will miss working with this great group of individuals.

Last, but certainly not least, I must thank my wife, Monica, for her invaluable love and support. Every time I began to doubt myself, she would be there to comfort me. I could not have completed this work without her.

Contents

1 Introduction **1**

1.1 The Transition from ILP to TLP 1

1.2 Overview of SCMP 3

1.2.1 The Node 3

1.2.2 Memory 4

1.2.3 The Network 5

1.2.4 Input/Output 6

1.3 Need for Applications 6

1.4 Thesis Overview 7

2 Related Work **9**

2.1 Single-Chip Multiprocessing 9

2.2 Application Considerations 12

3 A Programmer's Guide to SCMP **14**

3.1 Instruction Set 14

3.2 The Compiler, Assembler, and Linker . . . 18
3.3 SCMP Libraries . . . 19
3.4 The Simulation Environment . . . 23
3.5 Debugging . . . 27

4 Application Overview 28

4.1 Conjugate Gradient . . . 28
4.1.1 Background and Algorithm . . . 29
4.1.2 SCMP Implementation . . . 29
4.2 QR Decomposition . . . 32
4.2.1 Background and Algorithm . . . 33
4.2.2 SCMP Implementation . . . 35
4.3 JPEG . . . 36
4.3.1 Background and Algorithm . . . 37
4.3.2 SCMP Implementation . . . 41
4.4 Summary . . . 43

5 Results and Analysis 44

5.1 Overview of Metrics . . . 44
5.2 Conjugate Gradient . . . 47
5.2.1 Speedup Results . . . 47
5.2.2 MOPS Results . . . 49

5.3 QR Decomposition . . . 52
5.3.1 Speedup Results . . . 52
5.3.2 MOPS Results . . . 53
5.4 JPEG . . . 54
5.4.1 Speedup Results . . . 54
5.4.2 MOPS Results . . . 59
5.5 Summary . . . 62
6 Conclusions **63**
6.1 Future Work . . . 63
6.1.1 Application Profiling . . . 63
6.1.2 Source Level Debugging . . . 64
6.1.3 Parallelizing Compiler . . . 64
6.2 Summary . . . 65

List of Figures

1.1 64 Node SCMP System. 3

1.2 Block Diagram of SCMP Node. 4

2.1 Classifications of Benchmarks. 13

3.1 SCMP Code to Executable Transitions. 19

3.2 SendDataIntValue Example with Notification. 21

3.3 SendBlock Example with Notification. 22

3.4 X-Window SCMP Simulator with 64 Nodes. 24

3.5 SCMP Node Window. 25

4.1 Conjugate Gradient Sequential Algorithm Pseudocode. 30

4.2 Conjugate Gradient Parallel Algorithm Pseudocode. 32

4.3 SCMP QR Decomposition Pseudocode. 36

4.4 High Level Steps of the JPEG. 37

4.5 DCT Before and After Images. 38

4.6 Data Distribution of JPEG. 42

4.7 SCMP JPEG Encoder Pseudocode. 43

5.1 Conjugate Gradient Speedup. 48

5.2 Average Messages in Network during CG Execution. 48

5.3 CG MOPS Comparison using Rival Clock Speed. 51

5.4 QR Decomposition Speedup. 52

5.5 QR MOPS Comparison using Rival Clock Speed. 56

5.6 JPEG Speedup using Grayscale Images. 57

5.7 JPEG Speedup using Color Images. 58

5.8 JPEG MOPS Comparison using Rival Clock Speed. 61

List of Tables

3.1 Standard SCMP Instruction Set. 15

3.2 SCMP Special Registers. 16

3.3 Custom SCMP Instructions. 18

3.4 SCMP Available Standard C Libraries. 20

3.5 Simulator Configuration File Parameters. 26

4.1 Block Data Layout of a 128x4 Matrix on an 8x8 Processor Grid. 31

4.2 Cyclical Data Layout of a 128x4 Matrix on an 8x8 Processor Grid. 31

4.3 Householder Annihilation of Columns, m x n Matrix [19]. 34

4.4 Scatter Data Layout of a 4x4 Matrix on a 2x2 Processor Grid. 35

4.5 Typical JPEG Quantization Table [26]. 39

4.6 JPEG Zigzag Traversal Order [26]. 40

5.1 Test Systems Compared to SCMP. 45

5.2 Rules for Counting MOPS [25]. 46

5.3 OP Count of CG Algorithm, n x n Matrix with m Non-zero Elements. 49

5.4 MOPS Results using Conjugate Gradient. 50

5.5 OP Count of QR Algorithm, n x n Matrix, Householder Vector of Length k. 53

5.6 MOPS Results using QR Decomposition. 55

5.7 OP Count of JPEG. 59

5.8 MOPS Results using JPEG. 60

Chapter 1

Introduction

The advent of new architectures yields the inherent need for applications that showcase and benchmark them. The code of such programs is ideally standardized, but in some cases a custom algorithm must be implemented. This thesis presents the design and analysis of three high-performance applications for the Single-Chip Message-Passing (SCMP) Parallel Computer. The thesis begins with the motivation behind SCMP and a description of the architecture. The chapter also presents the need for applications and concludes with the thesis overview.

1.1 The Transition from ILP to TLP

Hardware designers continue to push the limits of microprocessors. By using techniques such as out-of-order execution, wide-issue pipelines, and branch prediction, computer architects are relentlessly seeking ways to speed execution times [34]. These performance improvements do not come without costs, however.

Design team sizes continue to grow while at the same time development times are not being sustained [6]. Indeed, manufacturers are spending more time engineering processors with a greater number of personnel at a higher cost. Research is predicting that by the year 2007, there will be over one billion transistors on a single chip [3]. Who will design, test, and verify a complex architecture of this caliber? The answer is even larger teams taking more time to develop an architecture will produce nominal performance gains over its predecessors.

Many, if not all, of these economic costs can be directly attributed to growing architectural barriers. As clock speeds continue to increase and transistor counts grow, researchers struggle with how to extend the use of a single clock to latch all devices on a chip. The increased transistor density yields a higher resistance between interconnects, and thus signal propagation is increased. Early evidence shows that long interconnect wires will limit maximum clock speed on future processors [5][29]. In addition to the growing latency problem, computer architects are faced with the task of finding new and beneficial ways to exploit instruction-level parallelism (ILP) within applications. This process is reaching a point of diminishing returns [5][6]. The involved complexity overhead is not becoming justifiable for the marginal performance increase due to capitalized ILP.

To combat these mounting architectural impediments, researchers are turning to thread-level parallelism (TLP) [15]. Investigation has shown that many applications contain significant amounts of TLP [5][6][17]. Furthermore, one solution to the signal propagation problem is to use chip multiprocessing (CMP), or multiple cores on a single die [5][6][15]. This technique may worsen single thread performance, but overall throughput can be improved. Another benefit of CMP is that development times can be reduced due to replicated cores across the chip [21].

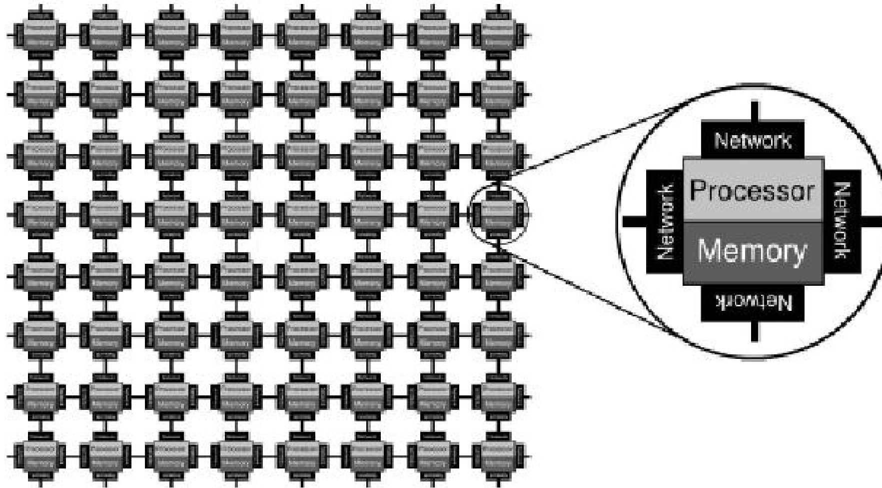


Figure 1.1: 64 Node SCMP System.

1.2 Overview of SCMP

To address the growing architectural and economic trends, the Single-Chip Message-Passing (SCMP) Parallel Computer was developed. The unique, tiled architecture attempts to exploit TLP, minimize wire traces, and decrease power consumption. Furthermore, by placing identical cores throughout the system, design costs can be reduced [5].

Figure 1.1 shows a high-level view of the SCMP system. With the use of a simulator, a 64-node configuration can be tested. Each processor contains up to 8 megabytes (MB) of memory and network connections to its north, south, east, and west neighbors are maintained.

1.2.1 The Node

Each node contains a 32-bit reduced instruction set computer (RISC) core, local memory, and registers. The CPU maintains both an arithmetic logic unit (ALU) and a floating

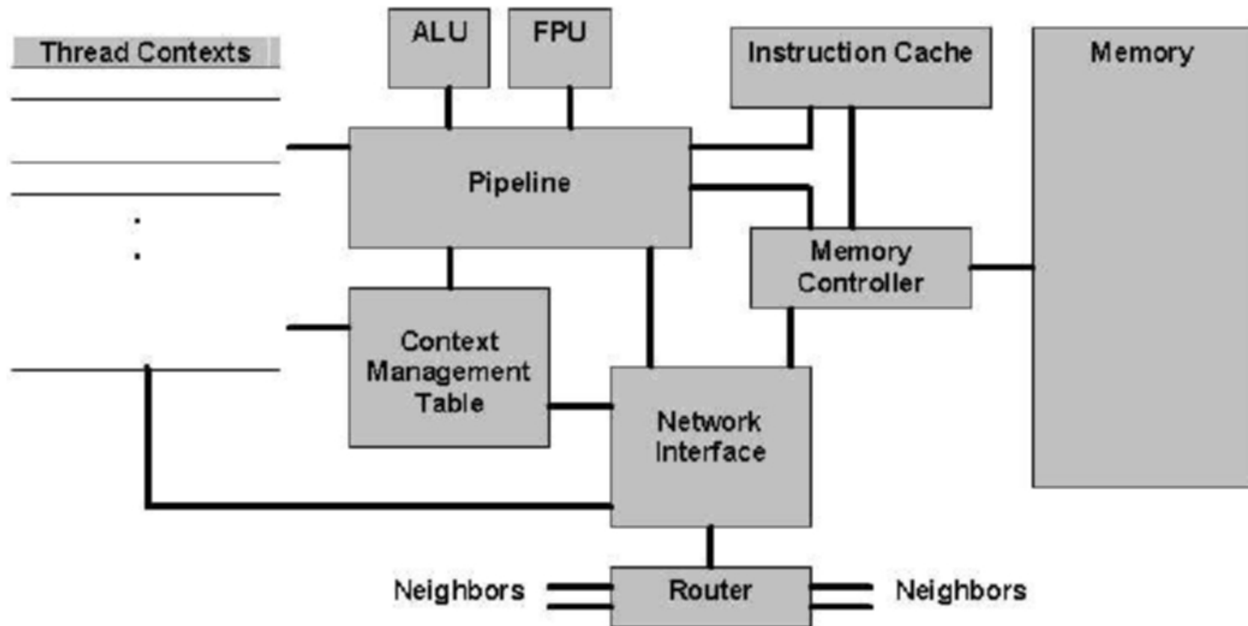


Figure 1.2: Block Diagram of SCMP Node.

point unit (FPU). The ALU and FPU are part of the 4-stage central processing unit (CPU) pipeline. An SCMP node also supports up to 16 threads of execution, each with its own set of 32 registers. The threads are scheduled in non-preemptive round-robin order in hardware. After a thread has begun, a context switch will not occur until it exits or suspends. An exception, network communication, or an explicit suspend are the only means by which a thread can halt. The large register file in combination with hardware-based scheduling, allow a context switch to occur in little as 3 to 4 clock cycles [5]. This functionality tends to offset the stringent scheduling mechanism. Figure 1.2 highlights the primary components of the SCMP node.

1.2.2 Memory

The local memory and its controller are important components in the SCMP system. The pipeline, instruction cache, and network interface unit (NIU) coordinate access to the local

memory through a custom controller. By avoiding shared and external memory schemes, all accesses are reduced to less than 3 clock cycles. If a node desires access to another's data, an explicit message to do so must be sent through the on-chip network. This scheme is further described in Section 3.3.

Because the memory is integrated on-chip with the CPU in SCMP, only a limited amount of memory is available for each processor. This limitation is an important one to note for the application developer. Writing programs for SCMP currently requires the programmer to partition the data in order to map the application SCMP grid. The choices made during this partitioning have a significant impact on performance.

1.2.3 The Network

The SCMP on-chip network is arranged as a 2-dimensional grid. The node components responsible for communication are the network interface unit (NIU) and router. The NIU can both send and receive messages as well as route flits (flow control digits) to its neighbors. SCMP routing is accomplished by using a wormhole algorithm in combination with virtual channels to reduce latency and prevent deadlocks [18].

Two types of messages, data and thread, are supported by the SCMP architecture. Within each message, a header flit contains the destination address of data, or alternatively, the entry point of a remote thread to be created. There is no receipt reply to the sender after data has arrived at the destination. Any required synchronization is left to the programmer. One technique sometimes used for this task is a callback routine that increments a count upon completion of a remote function. There is ongoing research to relieve the burden of synchronization from the programmer [30].

Thread messages are an important concept for the SCMP application developer to fully grasp. When a thread message arrives at a particular destination, a context is automatically

allocated for it by the NIU and its instruction pointer is set. If the NIU is unable to allocate a context for an incoming thread, the message will become blocked in the network. The message will remain blocked until a context is subsequently freed. The programmer should take precautions not to overwhelm a given node when implementing parallel algorithms.

1.2.4 Input/Output

Before the SCMP architecture can be fully utilized, it must include support for communication with external devices. Examples of possible components include storage devices, user input tools, and network cards. Although there is currently no support for such devices, the SCMP simulator provides mechanisms to access the keyboard, display, and filesystem of the underlying Linux operating system. Section 3.1 describes this process in more detail.

Several approaches have been proposed to answer the I/O problem of SCMP. One solution calls for each node to send its external requests to edge nodes which will in turn forward the request off-chip. Replies would then return to the sender in reverse order. Another design could allow a node to send external requests to a column agent and receive replies from a row agent. A third solution might be to give each node its own external access. The NIU could then be responsible for coordinating internal and off-chip requests. At present, no research has focused on this specific aspect of the SCMP architecture.

1.3 Need for Applications

The SCMP architecture is evolving from simulation to functional hardware. However, the performance of the unique design is still in question. Only a handful of applications for SCMP exist and even fewer allow comparisons to other architectures. Optimally, one would be able to extract an application from a given processor, recompile it for SCMP, and draw

comparisons based on the simulation data. Unfortunately, the SCMP interface does not afford the programmer this luxury. Each program must be either developed from the ground up or adapted to the SCMP message passing model. These reasons in combination with the lack of input and output, make performance comparisons to other architectures difficult. Until a standard chip-multiprocessing benchmark suite is adopted by computer architects, accurate performance comparisons to other parallel systems will remain problematic. This issue is discussed more in Chapter 2.

Given these limitations, this thesis attempts to highlight high-performance algorithms where the SCMP architecture excels. Three applications were developed: the conjugate gradient, QR decomposition, and JPEG encoder. These applications were used to compare the performance of SCMP to a number of other systems. The results show that the applications contain significant amounts of thread-level parallelism, and that SCMP is able to outperform other architectures that only exploit instruction-level parallelism. It is hoped that these parallel programs will provide future researchers insight as to how SCMP would best be utilized - whether it be in audio/video processing, digital signal processing, high-performance computing, or embedded, low-power applications.

1.4 Thesis Overview

Before discussing the details of the applications, it is important to understand the motivation for choosing them. Chapter 2 presents a review of several single-chip multiprocessors and focuses on the programs developed by others for each. The chapter concludes with application considerations for SCMP.

The programming formalities of SCMP are highlighted in Chapter 3 in addition to an introduction to the simulation environment. This background is essential for describing the implementation of the applications. These details, along with an overview of each sequential

algorithm, are presented in Chapter 4.

Chapter 5 provides the results obtained after testing each application on SCMP and four other systems. Finally, the thesis concludes by outlining areas for future work and a summary of the accomplished research in Chapter 6.

Chapter 2

Related Work

The idea of single chip multiprocessing or multiple cores on a single die is not an entirely new idea. This chapter presents a brief historical look at some of the single chip architectures and the applications, if any, available to them. The premise of the chapter is to provide insight into the CMP designer's desire for standardized programs to adequately benchmark the unique parallel systems.

2.1 Single-Chip Multiprocessing

The Briarcliff Multiprocessor was one of the first architectures aimed at exploiting fine grain parallelism with the use of multiple processors on a single chip [20]. The design contained four identical 36-bit RISC cores based on the Sun SPARC architecture. In addition, the Briarcliff Multiprocessor provided four memory banks accessible to each of the processors and was designed to primarily exploit instruction-level and loop level parallelism. This task, in addition to resource scheduling, was performed by a custom parallelizing compiler. The literature provides no references to applications or performance evaluations of the architec-

ture.

Another approach to single-chip multiprocessing was the MIT Reconfigurable Architecture Workstation (RAW) [39][40]. The RAW system was comprised of 16 RISC processors, each with 128KB of memory connected via a 2-D mesh grid. Speedup was achieved by capitalizing on streaming, instruction level, and data parallelism using a series of specialized compilers. Various benchmarks were designed for the RAW architecture including the Fast Fourier Transform (FFT) and the Data Encryption Standard (DES) [4]. The drawback of these applications was that they were written in Behavior Verilog and not easily ported to other systems. Third party programs developed using the C language were implemented for RAW, however. This allowed for comparisons to other architectures [39].

A predecessor to SCMP, the Pica system was aimed at high-throughput, low-memory applications [45]. Each node in the architecture contained less than 20KB of memory and data was processed using a 32-bit RISC core. Although no high-level compilers were used, several applications for the Pica system were developed using its assembly language. The suite included parallel matrix multiplication, jpeg compression, and thermal relaxation algorithms. The results showed that common applications could be adapted to low-memory requirement architectures. The Pica system was difficult to compare to other architectures because of the lack of hardware and standardized application code.

One of the pioneer single-chip architectures that targeted thread-level parallelism was the Hydra chip multiprocessor(CMP) [22]. Each chip contained four processors with local cache and access to a shared external memory. The idea behind the Hydra was to divide a given program in to speculative threads with a compiler. Using both hardware and software, the threads were divided among the four cores. Since the Hydra architecture provided an automatic parallelizing compiler, standard codes were able to be quickly ported to the system. Several of the Standard Performance Evaluation Corporation 1995 (SPEC95) benchmark suite applications were subsequently ported to Hydra and evaluated [23].

The RAPTOR architecture, similar to the Hydra CMP, contained four RISC processors on a single die and aimed to exploit thread-level parallelism [36]. However, the RAPTOR chip differed by providing a shared graphics co-processor and was based on the 64-bit SPARC version 9 instruction set. Each processor in the system accessed data via a shared external memory. The literature does not provide insight into how the architecture's performance was evaluated.

Online transaction processing was the workload target of the Piranha architecture. Using eight Alpha CPU cores and shared external memory, the system aimed to outperform conventional high-performance processors on commercial workloads [6]. The Piranha architecture was designed for scalability. In fact, input and output were done by a separate chip in the system. The design was evaluated using benchmarks based on those from the Transaction Processing Performance Council (TPPC) which involve the querying of large databases.

The Atlas CMP took a novel approach to the parallelization concept. Instead of having the user recompile code to extract parallelism, or worse, re-write code with explicit parallelism in mind, the Atlas architecture attempted to exploit parallelism within existing sequential binaries. Threads were dynamically extracted from the sequential code using a technique known as MEM-slicing [11]. The system contained up to eight processors, based on the Alpha architecture, connected via a pipelined, bidirectional ring. The MEM-slicing algorithm allowed applications to be easily migrated to the Atlas architecture. Performance of the system was evaluated using the SPEC95 benchmark suite.

Parallelizing compilers were again used for the speedup attained by the low-power Merlot [31] and the OSCAR CMPs [27]. Both designs placed four processing elements on a single chip. While the Merlot compiler focused on speculative threads, the OSCAR system exploited task-level, loop-level, and fine-grain parallelism.. The literature suggests that several programs were developed for the Merlot architecture, including a parallelized mpeg3 decoder. However, the basis and details of the applications were omitted. A jpeg encoder from the MediaBench

suite was implemented for the OSCAR CMP.

The Tera-op Reliable Intelligently adaptive Processing System (TRIPS) consisted of eight processors each with 64 available ALUs. This configuration combined with a custom scheduling compiler allowed the architecture to exploit thread, data, and instruction level parallelism [38]. The goal of TRIPS was to provide a highly adaptable system to a given environment or application. Furthermore, the chip sought to provide scalability with the design as technology and transistor counts increased. Several applications from various scientific domains were implemented for the TRIPS architecture including programs from the SPEC2000 benchmark suite [37][38]. The drawback is that all were hand-coded in TRIPS assembly language to take advantage of parallel hardware mechanisms. Thus, comparing the TRIPS system to other architectures remained difficult.

2.2 Application Considerations

As the previous section has shown, few common benchmarks exist among the single-chip computers. Much of the decision to implement a given application lies in the available memory to the architecture. Another factor to consider is the programming interface of the system. Also, if the programmer is left to extract parallelism from a given application rather than a parallelizing compiler, development time is increased considerably. Furthermore, if the original source code is changed, questions arise disputing whether the performance increase is due to the changed algorithm or the improved architecture.

Benchmarks can be classified into one of several categories. Figure 2.1 shows a breakdown of the common groupings. High performance benchmarks are typically reserved for server or desktop systems and assume a surplus of memory is available to the given architecture. Alternatively, application suites such as the Embedded Microprocessor Benchmark Consortium (EEMBC), develop programs aimed at embedded devices with limited resources [44].

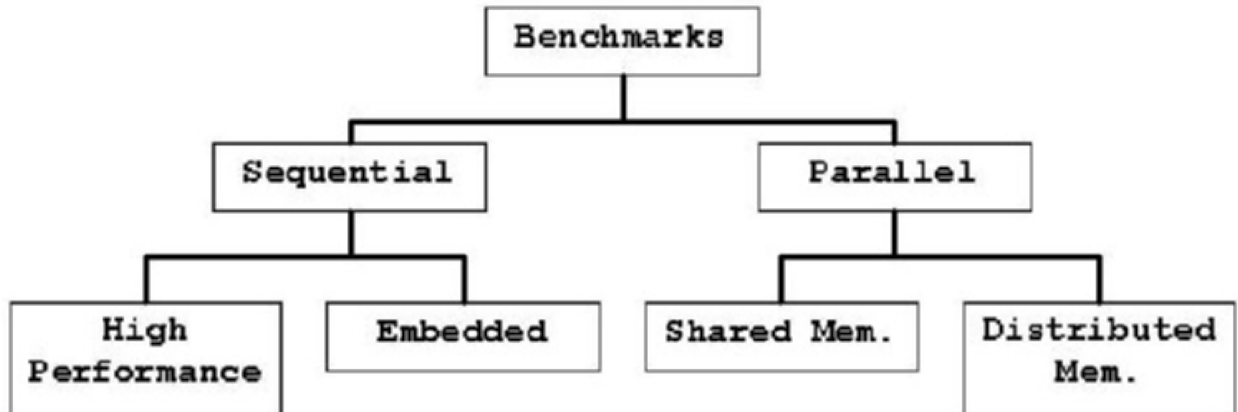


Figure 2.1: Classifications of Benchmarks.

Parallel benchmark sets are usually geared toward high-end clusters or supercomputers [46].

This scenario presents parallel embedded systems, such as SCMP, with difficult choices on how to evaluate performance. Given that the architecture of SCMP does not provide an automatic parallelizing compiler, the decision becomes even more challenging. One option is to obtain existing sequential code, parallelize it by hand, and port it to the system. Another alterative is to modify a parallel application designed for a higher end system and scale the problem so that it may be accomplished by SCMP. Finally, the designer can build an application from the ground up in hopes of highlighting architectural features and attempt to draw comparisons when possible. The latter is the decision adopted by this thesis.

Chapter 3

A Programmer's Guide to SCMP

Before discussing specific design considerations for SCMP applications, it is beneficial to examine how programming for the architecture differs from standard practices. This chapter is meant to give the reader an overall understanding of how applications are constructed, beginning from assembly code and proceeding to the final process of debugging.

3.1 Instruction Set

SCMP uses a reduced instruction set computer (RISC) architecture [34]. The RISC architecture of SCMP is of importance to the programmer for several reasons. First and foremost, given the limited debugging capabilities of SCMP currently available, it is paramount that the programmer be comfortable with SCMP assembly code to understand thread execution. Second, because messages are sent through the network via direct assembly calls, understanding the network requires an understanding of SCMP assembly language.

All non-floating point arithmetic and logical instructions are 32-bit operations. Additionally,

Table 3.1: Standard SCMP Instruction Set.

Type	Instructions
Arithmetic	ADD, ADDU, ADDI, ADDUI, SUB, SUBU, SUBI, SUBUI, MUL, MULI, MULU, MULUI, MULH, MULHI, MULHUI, IDIV, DIVI, IDIVU, IDIVUI, MOD, MODI
Logical	NEG, AND, ANDI, OR, ORI, XOR, XORI, LSH, LSHI, ASH, ASHI, ROT, ROTI, SLT, SLTI, SLTU, SLTUI, SGT, SGTI, SGTU, SGTUI
Branch	BRA, BEQ, BNE, NGT, NGE, BLT, BLE, BSR_R, BSR_I
Load/Store	LHI, LLO, LDB, LDBU, LDH, LDHU, LDW, STB, STH, STW
Floating Point	ADDS, ADDD, SUBS, SUBD, MULS, MULD, DIVS, DIVD, NEGS, NEGD, SLTS, SLTD, SGTS, SGTD, SEQS, SEQD, CVTSI, CVTIS, CVTDI, CVTID, CVTSD, CVTDS, LDS, LDD, STS, STD

most of these instructions provide a mechanism to use a register source and a signed 13-bit immediate offset. The branch instructions use a comparison to zero to test for validity. SCMP loads and stores can access memory by byte (8-bit), half-word (16-bit), and word (32-bit) [7]. To calculate addresses, a base-displacement scheme is used. The displacement is a signed 13-bit value. SCMP provides support for 32-bit and 64-bit floating point precision calculations. Conversion instructions allow the programmer to easily manipulate how arithmetic calculations are performed. Table 3.1 shows the contents of the standard RISC instruction set. Although these instructions are critical components for the programmer of SCMP, most differ only slightly from textbook RISC architectures [34].

In addition to the standard instructions, SCMP provides several custom instructions to interact with the unique single-chip multiprocessor architecture. It is beneficial to examine these unique instructions in further detail. However, before beginning this dissection, the

Table 3.2: SCMP Special Registers.

Register	Name	Description
CLKH, CLKL	Clock high, clock low	32-bit high and low parts of clock.
NIR	Node identification	Node number (0-63). Read only.
ATR	Active thread	Thread that is currently active. Read only.
FPCR	Floating point context	Context ID that active thread can access.
X_DIM	X-dimension	X-dimension of network grid. Read only.
Y_DIM	Y-dimension	Y-dimension of network grid. Read only.
MASK	Mask	Exception enable mask.
STATUS	Status	Bit-map to identify pending exceptions.
SIGNAL	Signal	Most recent exception.
EHANDLER	Exception handler	Address of exception handler.
ETHREAD	Exception thread	Context ID of exception thread.
IP	Instruction pointer	Address of current instruction.
MEM_SIZE	Memory Size	Amount of memory available. Read only.

registers on which the instructions operate must be declared and defined. These special registers along with a brief description of each are found in Table 3.2.

Having identified the distinctive registers of SCMP, the programmer can begin to examine the custom instructions available to manipulate them. These instructions are highlighted in Table 3.3. Aside from the thread and context instructions, each of the custom SCMP instructions are found only in startup or library code. That is, the compiler will not generate any of the aforementioned instructions. The *READSR* and *WRITESR* instructions are used for reading and writing the registers of Table 3.2, respectively.

The programmer should pay particular attention to the network instructions of Table 3.3. A message consists of a header flit, a number of data flits, and a tail flit. Instructions are

available to inject each component of a message into the network. To build a message, the user first sends a header flit and address flit using one of the *SENDH_X* instructions. The header flit tells the receiver what type of message it is (thread or data), and the address in memory where either the new data is stored or new thread is to begin execution. The second phase of the transmission involves one or more *SEND*, *SEND2*, or *SENDM* calls to transfer data from registers or memory. Finally, the user must complete a transaction by sending an end flit. This can be accomplished with the *SENDE* , *SEND2E*, or *SENDME* instructions. It is important for the programmer to note that issuing any of the network instructions may result in the current thread being suspended. This will occur if the flit is unable to be injected into the network. Notice that when the thread is again scheduled for execution, it will try to inject the flit again, repeating until it succeeds.

Another instruction of significance for the programmer to observe is *OSCALL*. Currently, how SCMP will perform its input and output operations is still under development. As a temporary solution, the *OSCALL* instruction was created. This instruction, which is only understood by the SCMP simulator, is used for operating system functionality. For example, standard C calls to fopen() are built using the *OSCALL* instruction. When the simulator encounters the opcode, it hands the task off to the underlying Linux operating system. The *OSCALL* instruction provides support for random number generation, file manipulation, and process control.

The final instruction-level mechanism that needs to be fully realized by the programmer is how floating point contexts are created. Floating point contexts are not automatically assigned to a given active thread. Rather, when an active thread encounters its first floating point instruction, an exception is generated. An exception handling routine, whose address is stored in the EHANDLER register, is then invoked. The exception handler then determines that a floating point exception occurred and allocates (via an *ALLOC* instruction) a new floating point context to the active thread. The active thread with which to associate the new context is determined from the ETHREAD register. Upon the thread's completion,

Table 3.3: Custom SCMP Instructions.

Type	Instructions
Control	READSR, WRITESR
Network	SENDH_I, SENDH_R, SEND, SEND2, SENDE, SEND2E, SENDM, SENDME
Thread	SUSPEND, END
Context	ALLOC, FREEC
OS	OSCALL

FREEC deallocates the claimed floating point context and *END* will free the remaining resources associated with the thread. Again, this scenario is an important concept for the programmer to grasp. If there are no free contexts for the exception handler to allocate to the requesting thread, the thread could suspend for several clock cycles before a context becomes available. To the application, this could appear as a temporary deadlock situation. The programmer should take extra precautions to avoid thread starvation of available floating point contexts.

3.2 The Compiler, Assembler, and Linker

Although the application programmer should rarely be concerned with how the specifics of object code generation occur, it can be useful knowledge in certain situations. This is true especially given the primitive debugging capabilities of SCMP. A brief overview of the SCMP development tools is presented below.

The compiler for SCMP supports compilation of the C language and is similar to the Gnu Compiler Collection (GCC). Like GCC, the SCMP compiler accomplishes its task in two

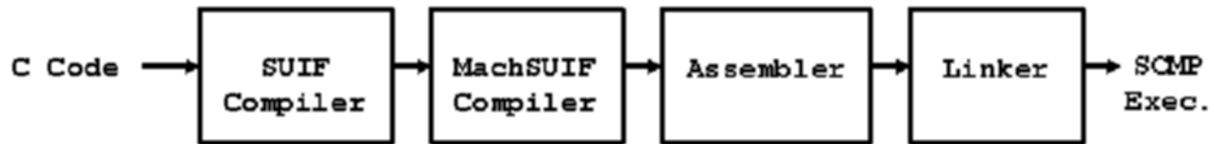


Figure 3.1: SCMP Code to Executable Transitions.

phases. The first step involves translating a high level programming language to into a platform neutral intermediate format. Examples of such high level concepts include loops, data structures, and function calls. SCMP selected to use the Stanford University Intermediate Format (SUIF) toolset for its high-level translation. After the C code is transformed to its intermediate format, a second pass must be performed to convert it to assembly code. The Machine SUIF (MachSUIF) toolset is used to perform this task. Finally, SCMP-specific optimizations can then be carried out to remove extraneous computations [7].

Both the assembler and linker were developed exclusively at Virginia Tech and are specific to the SCMP architecture. The assembler translates SCMP assembly code into object files which are passed to the linker. Finally, the linker combines one or more object files into a single executable which is understood only by the SCMP simulator. The overall process from C code to SCMP executable can be seen in Figure 3.1.

3.3 SCMP Libraries

The most important tools to the application programmer are the C libraries which allow one to take full advantage of the SCMP architecture. Two types of libraries have been developed, the Standard C libraries and the SCMP specific libraries. Some of the available Standard C libraries are summarized in Table 3.4. The SCMP libraries include routines for communication, thread management, synchronization, and access to the SCMP special

Table 3.4: SCMP Available Standard C Libraries.

File	Description
stdio.h	Input/Output with streams.
stdlib.h	General purpose - including memory allocation, process control, and conversions.
string.h	String manipulation.
time.h	Time and date related functions.
math.h	Mathematical calculations, with floating point support.

registers.

The SCMP libraries are essential for exploiting parallelism within applications. The two most basic functions for sending data from node to node are *SendDataIntValue* and *SendDataFloatValue*. These two functions send 32-bit integers and 64-bit doubles from one node to another, respectively. The only requirement of each is that the destination memory address on the receiver be known by the sender. (When a variable is globally declared in SCMP, it is mapped to the same memory location on every node in the system and thus known at compile time.) After the data is sent, there is no notification when it arrives at the destination. To accomplish this, the SCMP programmer will typically suspend the sending thread and wait on a the receipt of an acknowledgement from the receiver. An example of this mechanism is shown in Figure 3.2.

The next logical question is how to move blocks of data between nodes on the system. If memory is dynamically allocated on a processor, the address of it will not be known to other nodes. Therefore, the correct address of the memory must be explicitly passed from a receiver to a desired sender. Only after the transmitting node receives the destination address, can it invoke *SendDataBlock* to send a continuous (or strided) block of data to the recipient. The destination block address (on the receiver) is passed by the source typically

```
int num, sync;
int main()

  num = 16;

  /* initialize sync */
  sync = 0;

  dst = 1; /* destination of receiver */

  /* send value */
  SendDataIntValue(dst, &num, num);

  /* send sync */
  SendDataIntValue(dst, &sync, 1);

  /* wait on reply */
  while(sync == 0) suspendThread();
```

(a) Sender - Node 0

```
int num, sync, dst;
int main() {

  /* initialize sync */
  sync = 0;

  dst = 0; /* destination of sender */

  /* wait on sync */
  while(sync == 0) suspendThread();

  /* should now have value of num */

  /* send reply */
  SendDataIntValue(dst, &sync, 1);
```

(b) Receiver - Node 1

Figure 3.2: SendDataIntValue Example with Notification.

with a call to *createThread()*. This function is used to establish either a local or remote thread of execution. Again, there is no notification to the sender or receiver when the data block has arrived or completed. Normally, a receiver will reply to the sender as well as modify a local synchronization variable. Figure 3.3 outlines this scenario. An alternative for the programmer is to use the blocking *getBlock()* function which does not return until the requested data is received from the sender or to use a callback function which modifies a local counter.

In addition to single thread creation, there are several other calls unique to SCMP that prove useful in parallel applications. *parExecute* can be invoked from any node and allows the programmer to simultaneously execute multiple copies of a thread on all nodes in the system. This routine will not return until all nodes have completed the requested thread. The *parExecute* function can also return a value which allows the programmer, in effect, to perform a reduction. The *broadcastInt* and *broadcastFloat* routines transmit data values to

```
int* array;
int sync;
int main() {
  . . .
  sync = 0; /* initialize sync */
  . . .
  /* wait until data block sent */
  while(sync == 0) suspendThread();
  . . .
}

#pragma REMOTE
SendBlock(int **block addr, int size,
        int dst, int *dst_addr,
        int* sync_addr, int *tsync_addr)
{
  /* send the data, stride of 1*/
  sendDataBlock(dst, dst_addr, 1,
        *block_addr, 1, size);

  /* send notification */
  sendDataIntValue(dst, sync_addr,
        1);

  /* notify other local threads */
  *tsync_addr = 1;
}
```

(a) Sender - Node 0

```
int* array;
int sync;
int main() {
  int num_vals, myID;
  . . .
  sync = 0; /* initialize sync */
  . . .
  num_vals = sizeof(array);
  myID = getNir();

  /* get data from node 0 */
  createThread(0, SendBlock, NULL,
        &array, num_vals, myID, array,
        &sync, &sync)
  . . .
  /* wait for notification */
  while(sync == 0) suspendThread()
  . . .
}
```

(b) Receiver - Node 1

Figure 3.3: SendBlock Example with Notification.

all other nodes. No notification is returned upon receipt of the data and the task is again left to the programmer. The final function implemented for SCMP that is required of many parallel algorithms is *barrier*. Once initialized, any node that calls *barrier* will not proceed until all nodes in the system have invoked the routine. Careful planning should be followed when using calls to *parExecute* and *barrier*. The round robin thread scheduling of SCMP can severely limit system performance if these functions are used improperly.

Noticeably absent from the SCMP communication libraries are the simple sends and receives desired by most parallel application designers. But given the unique architecture of SCMP, this functionality is currently not possible. However, research is underway to investigate alternative means of delivering data across the on-chip network [30].

3.4 The Simulation Environment

The SCMP simulator is a unique piece of software in that it not only simulates the central processing unit (CPU), but it also gives an accurate depiction of the activity on interconnection network. The simulator is written for the Linux operating system and includes both a command-line access as well as X-Window graphical user interface (GUI). By using the GUI, the application developer can view the state of the system at any point in time during a program's execution. Figure 3.4 shows a screenshot of the main SCMP simulator window.

Each square in Figure 3.4 represents a given node in the SCMP system. Black squares represent inactive processors while green and red squares show active and stalled nodes, respectively. This allows the application programmer to identify possible "hot spots" in the network. By minimizing the number of stalled and inactive nodes, the developer is afforded another avenue to performance optimization.

The X-Window display also allows the application programmer to view specific information

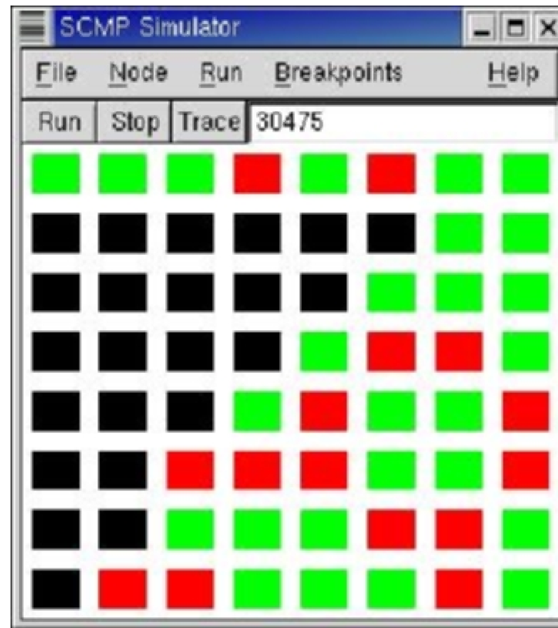


Figure 3.4: X-Window SCMP Simulator with 64 Nodes.

regarding a given node. By clicking on any square from the main display, the user can view the context management table, memory, registers, and other information specific to each node in the system. Each node window also contains its own standard output (stdout) display. This is important for debugging because multiple nodes may be printing to the terminal at any given time. The node window overcomes this handicap by printing only the characters output its respective processor. Figure 3.5 shows an example node window.

The final advantage that the SCMP simulator affords is one of versatility. With the use of a configuration file, the application programmer is permitted to change a variety of system parameters including, the number of nodes in the system, instruction cache size and parameters, memory size, and number of contexts per node. The configuration file also can provide valuable metering information, if so desired. Table 3.5 shows the available options in the configuration file along with their default values.

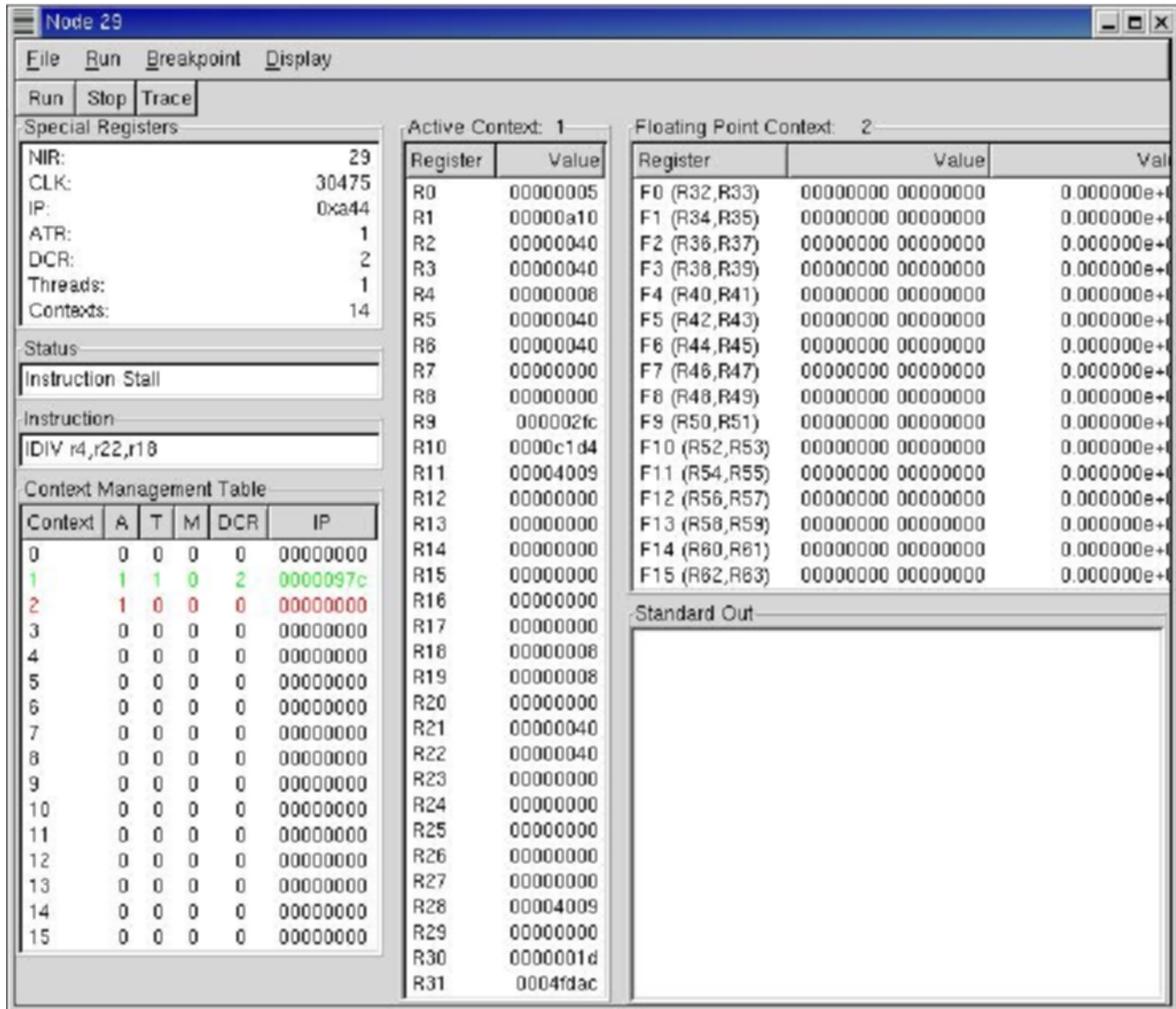


Figure 3.5: SCMP Node Window.

Table 3.5: Simulator Configuration File Parameters.

Parameter	Description	Default
X	Num. nodes x-dimension	8
Y	Num. nodes y-dimension	8
Mem_Size	Memory per node, bytes	1,048,576 (1MB)
Num_Contexts	Max threads per node	16
I_N	Num. sets instr. cache	32
I_K	Num. lines per set instr. cache	1
I_L	Num. words per line instr. cache	8
meter_interval	Num. clocks between log file updates	1000
meter_log_file	Execution statistics	n/a
icache_file	Instr. cache statistics	n/a
concurrency_file	Num. active nodes per update interval	n/a
message_file	Message statistics	n/a
network_file	Network statistics	n/a
message_traffic_file	Global message trace	n/a

3.5 Debugging

Currently, the debugging environment for the SCMP architecture is still in its infancy stages. The SCMP simulator will generally output the address on which a given error occurred. The application developer must then generate a memory map of the executable using the SCMP linker to determine the exact line number of the failure. Future work that will greatly benefit the programmer will include support for source level debugging similar to the GNU debugger (GDB). This addition will require support from both the compiler and simulator, as well as possible development of another standalone tool.

More important to the application developer would be the implementation of an executable profiler. This tool would be responsible for collecting information such as the number of clock cycles spent in a function and percentage of execution spent in a function and its subroutines. The profiler would allow the programmer to determine where the majority of clock cycles were being spent and how overall performance could be improved. The SCMP architecture presents a unique problem to the implementation of a profiler because of its multithreaded, multiprocessor design. The realization of such a tool would require significant changes to both the compiler and simulator.

Chapter 4

Application Overview

This chapter presents three high-performance applications for the SCMP architecture. The applications were chosen for various reasons. The JPEG encoder was selected to determine SCMP's performance in an image processing capacity. The conjugate gradient and QR decomposition programs were implemented to further understand the architecture's ability to manipulate large data sets. All three applications give the designer insight into how the on-chip network affects overall system throughput.

4.1 Conjugate Gradient

The conjugate gradient (CG) application was derived from the matrix stressmark of the Data-Intensive Systems (DIS) Benchmark Suite [41]. The DIS suite provides pseudocode and thorough description of each of its algorithms. However, the pseudocode does not specify where or how parallelism exists in the CG. This task is left to the implementer. An overview of the conjugate gradient algorithm is presented below. The section concludes by outlining the parallel SCMP implementation.

4.1.1 Background and Algorithm

The CG is commonly used to solve both nonlinear and linear systems of equations. Furthermore, the literature shows that signal processing and image restoration applications make use of the conjugate gradient extensively [16][32]. In addition, benchmark suites such as DIS employ the CG because it accesses data in mixed patterns [41]. The conjugate gradient was implemented for SCMP in an attempt to compare the architecture with other parallel systems. The CG algorithm solves equation 4.1 for x.

$$Ax = b \tag{4.1}$$

where A is a sparse n x n matrix, b is a 1 x n vector, and x is an unknown 1 x n vector.

By performing a series of iterations, estimates of x are calculated until Ax is within a certain tolerance of b. A unique solution x is guaranteed to exist is long as A is symmetric positive definite [19][41]. The DIS suite gives a description on how to randomly generate A to ensure a unique solution can be reached. With the use of a sparse matrix, large data sets are able to be computed. The pseudocode of the sequential CQ algorithm minus the matrix generation is shown in Figure 4.1.

4.1.2 SCMP Implementation

In a sparse matrix, only the non-zero elements need to be stored. For this algorithm, the sparse matrix A was implemented by using two dynamic arrays for each row. The first array contained the column offset of the entry while the value was found in the second. After the sparse matrix representation was completed, the next task was to determine how to best distribute the data across the nodes to both maximize computation and reduce

```
X = zerovector()
vectorR = vectorB
vectorP = vectorR

error = 1.0
while(error < errorTolerance) do

    alpha = (vectorR^t * vectorR) /
            (vectorP^t * (matrixA * vectorP))
    nextVectorR = vectorR - alpha * (matrixA * vectorP)
    beta = (nextVectorR^t * nextVectorR) /
           (vectorR^t * vectorR)
    vectorX = vectorX + alpha * vectorP
    vectorP = nextVectorR + beta * vectorP
    vectorR = nextVectorR
    error = |matrixA * vectorX - vectorB| / |vectorB|
```

Figure 4.1: Conjugate Gradient Sequential Algorithm Pseudocode.

communication. Two methods were subsequently tried. The first attempt used a block data layout outlined by [12] which is shown in Table 4.1. Testing revealed that the randomly generated values became clustered in the upper rows of the matrix. Therefore, the data distribution among nodes was not balanced which led to uneven computation in the system. A second method was tried in which matrix A was distributed in a cyclical order as shown in Table 4.2. This layout improved the computational throughout without increasing the communication overhead. This same distribution was also applied to vector b. To reduce communication, each node was given a complete copy the vectors x and p.

The SCMP parallel conjugate gradient algorithm is centered on balanced matrix-vector multiplication. The distribution of A and the fact that all nodes had x and p, allowed the Ax and Ap calculations to be performed entirely in parallel. Furthermore, all nodes duplicated the calculation of x since they already had the needed data. These redundant computations were faster than broadcasting a resultant vector from one or more nodes. The updating of p was more complicated, however. Since nodes could only update some elements of p (depending on elements of r), each node was required to update its "owned" values on every node in the system. But rather than iterating through each node and performing a broadcast to all, the communication was broken into two steps. Updates of p were first sent across each row of processor grid, then the same operation occurred among the colums. This design greatly

Table 4.1: Block Data Layout of a 128x4 Matrix on an 8x8 Processor Grid.

0	0	0	0
0	0	0	0
..	..	..	..
32	32	32	32
32	32	32	32
33	33	33	33
..	..	..	..
63	63	63	63

Table 4.2: Cyclical Data Layout of a 128x4 Matrix on an 8x8 Processor Grid.

0	0	0	0
1	1	1	1
..	..	..	..
63	63	63	63
0	0	0	0
1	1	1	1
..	..	..	..
63	63	63	63

```
rr = parExecute(vectorR = vectorB)
parExecute(vectorP = vectorR)
broadcast(vectorP)
error = 1.0
while(error < errorTolerance) do

    pAp = parExecute((vectorP^t * (matrixA * vectorP))
    alpha = rr / pAp
    parExecute(updateX)
    error = parExecute(CalcAx)
    error = sqrt(error) / |vectorB|

    if(error > errorTolerance)
        rr_next = parExecute(updateR)
        beta = rr_next / rr
        rr = rr_next
        parExecute(updateP)
        broadcast(vectorP)
```

Figure 4.2: Conjugate Gradient Parallel Algorithm Pseudocode.

improved the throughput and overall execution time of the algorithm. Figure 4.2 shows the pseudocode of the parallel CG algorithm for SCMP. Notice that since the result of the calculation is tested during every iteration, there is no need to explicitly check the output for accuracy.

4.2 QR Decomposition

One of the motivations for implementing the QR decomposition was to determine if SCMP's high speed on-chip network provided an additional benefit for the computation. Another goal was to investigate whether a multithreaded design could be applied to the QR decomposition. A brief outline of the basic algorithm and an overview of the SCMP implementation are presented below.

4.2.1 Background and Algorithm

The QR decomposition is used in many scientific engineering applications, including signal processing and least squares problems [2][24][47]. Furthermore, the algorithm has been included in multiple benchmarking suites [1][9]. The QR decomposition is used to solve equation 4.2 and is amenable to parallelization [8][14].

$$A = QR \tag{4.2}$$

where A is an m by n matrix, Q is an orthogonal m by m matrix, and R is an m by n upper-right triangular matrix.

There have been several methods developed to solve equation 4.2 such as Gram-Schmidt, Givens rotations, and Householder transformations [19]. Each of the approaches have been studied extensively. The Givens and Householder methods have been shown to adapt well to parallel architectures [47]. Typically, the Givens method is better suited for shared memory machines while the Householder algorithm integrates more easily with distributed memory architectures. Indeed, the programmer's choice of algorithm is heavily dependent upon the architecture for which it was implemented [47]. Given the memory configuration of SCMP, it follows that the Householder method is the algorithm of choice.

The Householder method itself has three variations: standard, row/column, and blocked. The standard method is designed as a sequential algorithm and is not intended to be executed in parallel. The row/column procedure involves the way updates take place across a given processor configuration. Finally, the blocked scheme is designed to be cache friendly because blocks are matched to the cache size of a given architecture. However, SCMP contains no data cache, so there is no advantage for using this method. Therefore, the row/column algorithm is used for this thesis.

Table 4.3: Householder Annihilation of Columns, m x n Matrix [19].

1st iteration	2nd iteration	3rd iteration	..	nth iteration
x	x	x	..	x
0	x	x	..	x
0	0	x	..	x
..	..	..	..	..
0	0	0	..	x

The Householder transformation operates by annihilating each of the subdiagonal elements of A over n iterations as shown in Table 4.3. Each stage involves the formation of a Householder matrix which is applied to a submatrix of A with the first element on the diagonal. However, the explicit formation of the Householder matrix can be avoided by dividing each iteration into the two steps shown in equations 4.3 and 4.4[47].

$$z^t = v^t A \tag{4.3}$$

$$R = A - \beta v z^t \tag{4.4}$$

where v is a Householder vector and $\beta = 2/v^t v$.

The results of the QR decomposition can be checked in one of two ways. The first is to use the output matrices Q and R in equation 4.2 and verify the result equals the input matrix, A. Another option given that matrix Q is orthogonal, one can check the result using equation 4.5.

Table 4.4: Scatter Data Layout of a 4x4 Matrix on a 2x2 Processor Grid.

0	1	0	1
2	3	2	3
0	1	0	1
2	3	2	3

$$I = QQ^t \tag{4.5}$$

where I is the identity matrix.

4.2.2 SCMP Implementation

The predominant task of the SCMP QR implementation was to determine how the input matrix was to be distributed among each of the nodes. Two layouts were subsequently implemented and tested. The first configuration allotted each node $\lceil n/64 \rceil$ columns of A. This arrangement, while easy to implement, suffered from load balance issues and failed to produce noticeable speedup. A second design based on the scatter data layout described by [12] was used in order to uniformly distribute the matrix across the grid of processors. The scatter layout ensured that each node remained active throughout the execution. Table 4.4 shows an example of the scatter data configuration.

The scatter layout dictated that both broadcast and reduction functions would need to be implemented in order to construct the necessary distributed vectors. To form the Householder vector, v, a reduction was performed by the owners of the current column in question. After calculating v, the owner would then broadcast the updated vector to the nodes in its column of the processor grid. Following the vertical broadcast, another horizontal update

```
For i to columns
    If(OwnColumn)
        V = Housevector()
        Barrier()
    Else
        Barrier()
        GetV()
    Barrier()
    CalcOwnedZ()
    Barrier()
    TreeReduction(Z)
    CalcR()
```

Figure 4.3: SCMP QR Decomposition Pseudocode.

by the column nodes provided the Householder vector to all nodes of the system.

A copy of v allowed all nodes to perform Equation 4.3 in parallel. However, this operation only provided a partial update of z. To complete the calculation, a tree reduction was performed simultaneously by each node. After obtaining the final value of z, Equation 4.4 was performed in parallel. Barriers were used between vector updates to ensure each node had received the broadcast data. The pseudocode for the formation of the Householder vector is found in [19] and final algorithm of the SCMP implementation shown in Figure 4.3.

4.3 JPEG

Since the adoption of the Joint Photographic Expert Group (JPEG) still image format in 1992 [26], researchers and scientists have strived to find faster, more efficient means of performing the computations necessary to encode raw image data. Even though network bandwidths continue to increase and secondary storages grow exponentially, there will always be a need for high performance image compression.

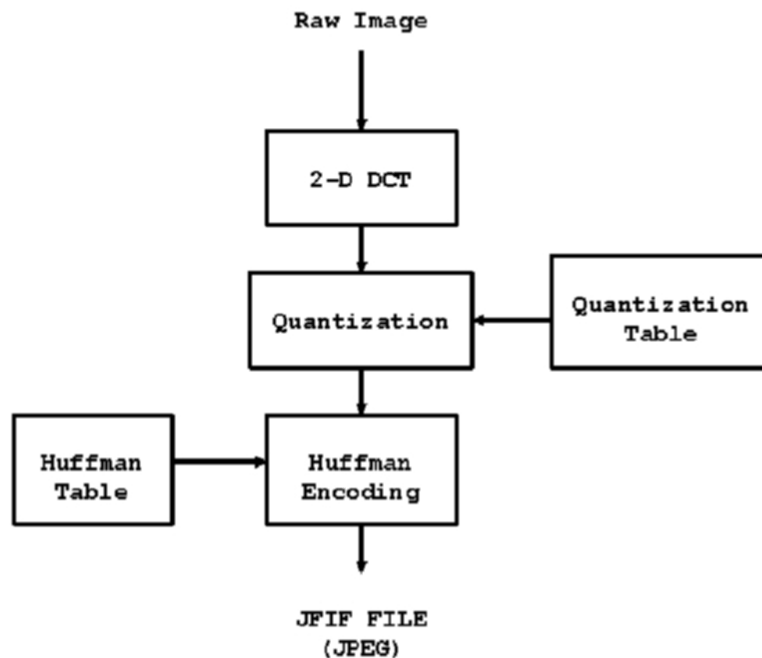


Figure 4.4: High Level Steps of the JPEG.

The independent steps of the JPEG algorithm mesh well with parallel architectures. Because blocks (groups of 8x8 pixels) can be compressed simultaneously, many different approaches to parallelization within hardware and software have been realized. The unique architecture of the SCMP lends itself well to a high performance implementation.

4.3.1 Background and Algorithm

The JPEG sought to create an image format that could handle high compression rates without a noticeable loss in quality. This was accomplished with the introduction of the JFIF file format. Although four modes of compression are specified, the most commonly used is sequential encoding; hence the algorithm will be described using this assumption. The high level steps of the method are shown in Figure 4.4.

Figure 4.5: DCT Before and After Images.

Raw images, usually containing one to three bytes per pixel (grayscale or color), are used as input into the JPEG encoder. Before entering the first step of the encoding process, an image is broken up into blocks of 8x8 pixels. The block is the basic computational unit of the JPEG. After an image has been subdivided, the two-dimensional discrete cosine transform (DCT) is computed over all blocks of the image. The 2-D DCT is outlined in equation 4.6 [43] and has been optimized in [10].

$$F(u,v) = \frac{1}{4}C(u)C(v)\left(\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\frac{(2x+1)u\pi}{16}\frac{(2y+1)v\pi}{16}\right) \tag{4.6}$$

$C(u), C(v) = \frac{1}{\sqrt{2}}$ for u,v = 0; $C(u), C(v) = 1$ otherwise.

The DCT converts an image from the spectral domain to the frequency domain. Then, each block contains sixty-three AC components and one DC component. The DC component is located at position (0,0) of the 8x8 matrix and typically contains most of the overall image data of the block. The remaining sixty-three AC components comprise the finer details of the image. The DCT puts the image into a format that can be easily compressed, but itself is a lossless transformation. Figure 4.5 shows an input and output image of the DCT.

The second phase of the JPEG encoding algorithm seeks to compress an image by dividing each data element by a quantization value. Different quantization values can be used to

Table 4.5: Typical JPEG Quantization Table [26].

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	69	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

selectively emphasize or de-emphasize different spectral components of the block. Many spectral components will be reduced to zero by this quantization, and will not need to be stored in the final output file which provides for much of the compression. As with the DCT, this operation is performed on a per block basis. The user can specify a custom quantization table in order to balance a filesize/quality tradeoff. However, [26] states that Table 4.5 has been shown to work effectively.

After quantizing the sixty-four values of the block, the final step of the JPEG is to encode the data. [26] specifies two types of encoding: Huffman and arithmetic. However most implementations use only the former, hence so does this thesis. It is important to note that many of the AC components in a given block are reduced to zeroes following quantization, hence the desire for run-length encoding. The number of zeroes preceding an AC value is termed the run, while the length is defined as the minimum number of bits required to represent the value. These run-length codes are defined in a Huffman table embedded within the header identified by the JFIF file format. [26] defines tables that have been shown to mesh well with the JPEG. Since most of the low frequency AC components are focused in the upper left corner of the block, a zigzag re-ordering of the sixty-three values is used. This block traversal is important to yielding a higher compression ratio. The block traversal order

Table 4.6: JPEG Zigzag Traversal Order [26].

n/a	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

is shown in Table 4.6.

Recall that the DC component is the first value of the block even after zigzag reordering. Since a value could never precede it, no run length is used to encode the DC component. However, because DC components do not vary significantly from block to block, only the difference in successive DC values is encoded. As one may infer, DC components must be propagated from block to block in order to construct the JPEG bit stream. Also, by only encoding the difference of successive DC values, another level of compression has been performed. However, this introduces a dependence between the blocks, and thus limits a parallel implementation. The DC value for a block cannot be encoded until the DC value for the previous block has been determined, which imposes a sequential ordering on the blocks.

One can avoid computing the difference between successive DC components with the use of restart markers. Restart markers, defined by [26], are a two-byte value placed in the bitstream between Huffman encoded blocks. If the preceding block does not end byte aligned, the remaining bits are padded with ones and the restart marker is inserted. The result of restart markers is a slightly lower compression ratio because of added data of the tag itself and the encoded DC value rather than the successive difference. However, this slight increase in block size is still negligible compared with the overall file size and is offset by the increased

parallelism available.

It has been shown that the DCT and quantization steps of the JPEG can be computed entirely in parallel. In addition, with the use of restart markers, [13] and others have shown that Huffman encoding can be independently computed. Therefore, the only required serial component of algorithm is the outputting of successive blocks. This serial limitation can be quantified by Amdahl's law shown in equation 4.7. It is evident that no matter how many processors are applied to the JPEG, the limiting factor will be the speed of outputting the data.

$$speedup_{max} = \frac{1}{F + \frac{1-F}{P}} \tag{4.7}$$

where F = percentage of time outputting and P = number of processors

One technique that has been used to overcome this output serialization is to use a parallel disk array as in [13]. Another approach to overcoming Amdahl's Law has been to pipeline the JPEG. That is, rather than performing all steps of the algorithm (DCT, quantization, and Huffman encoding), a processor or processing element is responsible only for computing a small portion of the JPEG. This leads to smaller data blocks written to disk more often, thus computational throughput is increased. Variations on parallelization and pipelining techniques have been used extensively throughout the literature [28][33][35].

4.3.2 SCMP Implementation

The SCMP architecture lends itself well to a high performance implementation of the JPEG. Because of the inherent thread-level parallelism (TLP) programming model, each of the obvious independent tasks (DCT, quantization) can be divided among the respective nodes

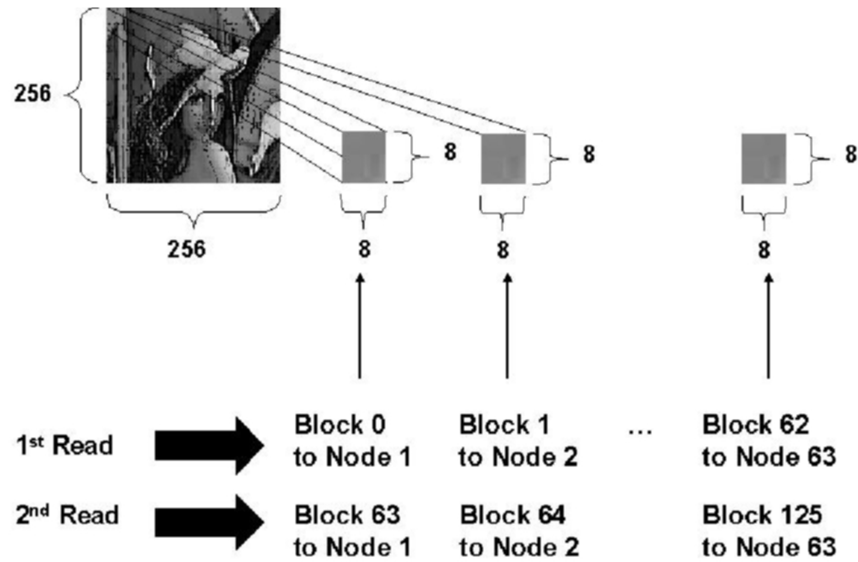


Figure 4.6: Data Distribution of JPEG.

in the system. In addition, with the use of restart markers, the Huffman encoding step can be performed void of inter-node communication.

The quantization and Huffman encoding computations are well defined and are not amenable to algorithm tweaking. However, a fast implementation of the DCT exists in [10] and was constructed for SCMP. To further reduce the computation time and control the execution, the DCT algorithm was implemented in assembly language rather than C code.

One of the difficulties faced in the SCMP implementation was to determine how the data was to be distributed among each of the sixty-four tiles. Each node was given a successive block of the input image and subsequent blocks were attained in a round robin fashion. To take a advantage of the high-speed interconnect, node zero was deemed responsible for requesting and writing the Huffman encoded blocks to disk. This data distribution can be seen in Figure 4.6.

The challenge was to provide enough work to the system so that node zero did not impede the algorithm by writing to disk. Ideally, node zero would request a block immediately following

```
parExecute(ReadImage)
Node zero: Write JPEG header
For i to blocks in image
     Node zero: request block i from owner
     Owner: compute DCT, quantization, and encoding of i
     Owner: send block i to Node zero
     Node zero: write encoded block i
     Node zero: write restart marker
```

Figure 4.7: SCMP JPEG Encoder Pseudocode.

the owner completing the Huffman encoding of that block. This approach would, in theory, lead to minimal thread suspension time. Two implementations were constructed and tested. The first called for each node (excluding node zero) to precompute the DCT and quantization of all owned blocks. Then, upon request from node zero, complete the Huffman encoding of the requested block. After testing the implementation on several images, it was determined that a given node would suspend several thousand clock cycles between node zero requests. For this reason, a second, final implementation was constructed to increase the granularity between block requests. The design called for each node to compute the DCT, quantization, and Huffman encoding of its blocks only after a node zero request. This design greatly improved the computational throughput. The pseudocode for the final implementation is shown in Figure 4.7.

4.4 Summary

This chapter has presented the sequential algorithms and parallel pseudocode for three high-performance applications that I implemented for SCMP. The conjugate gradient, QR decomposition, and JPEG were selected for various reasons, but all succeed in measuring the performance the of the system. As the next chapter will show, SCMP performs well when compared with other current architectures.

Chapter 5

Results and Analysis

Two performance metrics used to evaluate the SCMP architecture are presented in this chapter. The chapter begins by outlining the motivation and details behind each of the metrics. The last three sections show the results obtained using the implemented applications.

5.1 Overview of Metrics

The first metric used for evaluation was speedup. Speedup compares the performance of a single processor with that of multiple processors and is quantified in Equation 5.1 [15]. Given that the SCMP simulator allows for varying processor configurations, the metric provides a good indication of the benefit of the on-chip network and the scalability of the architecture and the algorithm. The problem size of each application was held constant and simulated with one to sixty-four nodes. The number of clock cycles required to complete each algorithm was used in the calculation.

Table 5.1: Test Systems Compared to SCMP.

Processor	Clock Speed(s)	Memory	OS	Num. Nodes
Intel Pentium 4	2.53GHz	1GB	Linux	N/A
IBM PowerPC G4	700MHz	512MB	OSX	N/A
Sun SPARC v9	500MHz	2GB	Solaris	N/A
Sun SPARC v9 cluster	(2) 500MHz, 400Mhz, 333Mhz	$\geq$ 512MB	Solaris	4

$$speedup = \frac{T_{1\,processor}}{T_{N\,processors}} \tag{5.1}$$

where $T =$ number of clock cycles.

Although the speedup metric shows the benefit of multiple SCMP processors over a single node configuration, it does not allow the performance of the architecture to be measured against other systems. However, comparison to other architectures is desirable to further evaluate SCMP. Before deciding what metric to use, the architectures with which to compare needed to be determined. Table 5.1 shows the four systems that were selected to measure against the SCMP architecture. These systems were chosen primarily because of their availability.

Since a sequential version of each application was already developed for the speedup metric, porting the code to the respective architectures involved little more than fixing endian issues [34]. But in order to make the parallel versions of the programs run on the cluster, the code had to modified to use the Message Passing Interface (MPI) [42]. This presented several design challenges as the MPI standard does not provide the functionality of the SCMP *parExecute* routine and others. Furthermore, the fact that SCMP communication is receiver-initiated and MPI is sender-receiver based, lead to several portions of code being

Table 5.2: Rules for Counting MOPS [25].

Operation	Count	Comment
Add, Subtract, Multiply	1	
Isolated Assignment	1	E.g. a = b
Comparison	1	
Type Conversion	1	E.g. C-style cast
Division, Square Root	4	
Sine, Exponential, etc	8	

modified extensively. Still, the MPI code was developed to match the SCMP code as closely as possible.

Ideally, the metric of choice to compare the systems of Table 5.1 to SCMP would be execution time. However, the execution time is essentially meaningless unless one is familiar with the underlying application. Another alternative metric that can be beneficial is instruction count. But since the group of systems to measure includes both RISC and Complex Instruction Set Computer (CISC) architectures, this comparison would also be flawed. The approach that was used to combine these two metrics, and relate to the application, was the calculation of MOPS, or millions of operations per second [25]. This technique allows comparisons to be made based on how fast a given algorithm is performed rather than focusing on the code, compiler, memory hierarchy, or other architecture specific features. [25] gives rules for counting an algorithm's operations which are shown in Table 5.2.

Given the current development state of SCMP, the exact clock speed is unknown. Therefore, a range of SCMP clock speeds is reported in the MOPS results. Best case estimates have shown that a 5GHz+ clock may be possible. However, future work could involve managing a clock speed versus power usage tradeoff. Hence, performance slower speeds is also reported in this thesis.

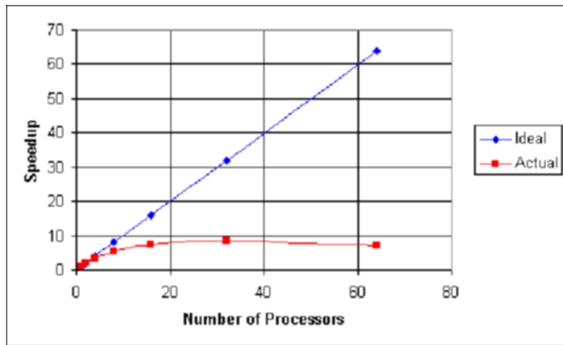
5.2 Conjugate Gradient

5.2.1 Speedup Results

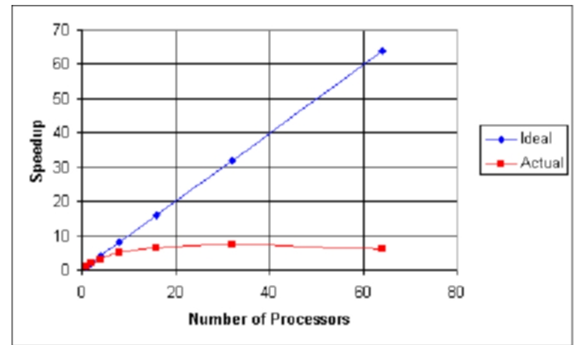
The conjugate gradient application is interesting to examine because the data distribution yields that several broadcasts and reductions be performed in the main execution loop of the algorithm. This communication provides insight on the impact of the on-chip network. To calculate the speedup, the CG program was simulated with one, two, four, eight, sixteen, thirty-two, and sixty-four node configurations. The results are shown in Figure 5.1.

Figure 5.1 shows that as the problem size increased, the speedup improved only marginally. The metric also revealed that speedup began to level off as the number of nodes exceeded four. To understand the observed behavior in more detail, two of the metering options of the simulator were enabled and analyzed. The concurrency log showed that throughout the execution of the CG with sixty-four nodes, on average about 42% were active at any given time. In comparison, the simulation with four nodes showed that 88% were busy. This discrepancy can be explained by examining the message log files of each configuration.

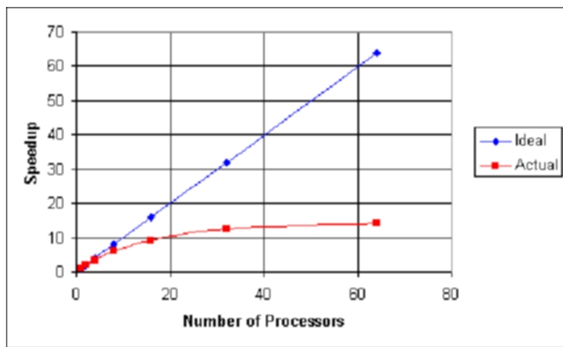
Figure 5.2, derived from the message log, shows the average number of messages in the network over the entire execution of the CG algorithm with four and sixty-four nodes. The sixty-four node simulation contained a higher number of messages in transit per interval than the former. This is due to the broadcast of the P vector and the three *parExecute* reductions of each iteration. Also, given that the problem was further subdivided in the larger processor configuration, the amount of work per node was also lessened. This leads to the conclusion that the communication to computation ratio was simply not high enough to sustain linear speedup as the number of nodes increased.



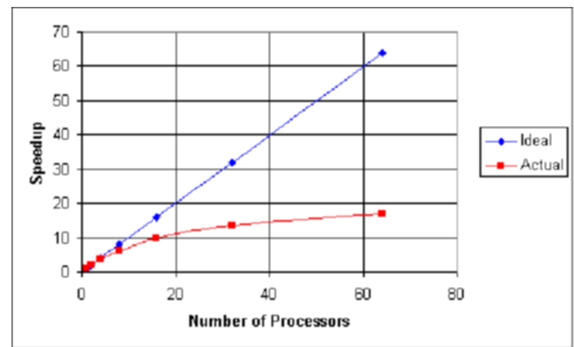
(a) 500 x 500 Matrix



(b) 1000 x 1000 Matrix

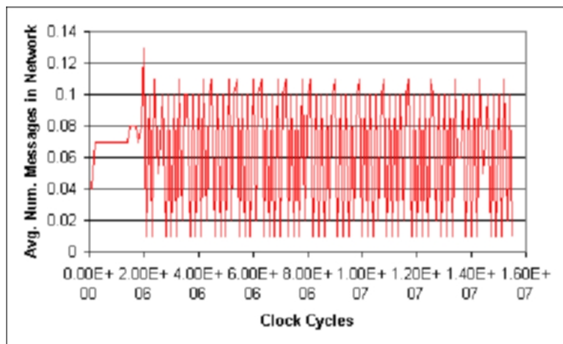


(c) 5000 x 5000 Matrix

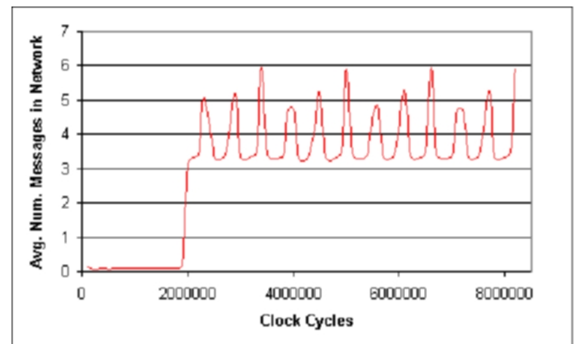


(d) 10000 x 10000 Matrix

Figure 5.1: Conjugate Gradient Speedup.



(a) 4 node configuration



(b) 64 node configuration

Figure 5.2: Average Messages in Network during CG Execution.

Table 5.3: OP Count of CG Algorithm, n x n Matrix with m Non-zero Elements.

	Operation	Count
	pAp = p^t * (A * p)	2m + 2n
	alpha = rr / pAp	1
	x = x + alpha * p	2n
	error = |A * x - b| / |b|	2m + 3n + 1
	r = r - alpha * A * p^t	2n
	rr_next = r * r^t	2n
	beta = rr_next / rr	1
	p = r + beta * p	2n
OPS per iteration		4m + 13n + 12

5.2.2 MOPS Results

In order to calculate the MOPS metric, the CG algorithm needs to be further examined. One cannot simply use the pseudocode of Figure 4.1 or Figure 4.2 to determine the number of OPS because they contain implementation details that are not essential to solve the problem. Rather, the necessary steps need to be outlined and the operations based on the size and elements of the input. These required steps of the CG algorithm with respective operation counts are shown in Table 5.3. By measuring the average time per iteration in several systems, the table allows one to make comparisons based on varying SCMP clock speeds.

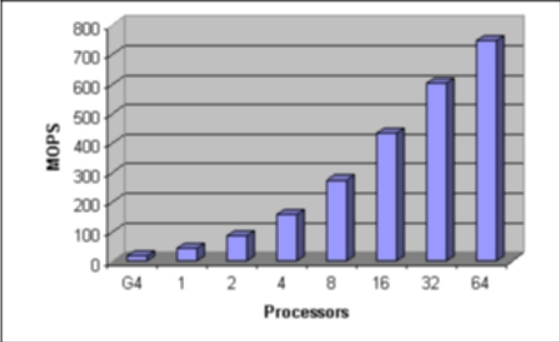
Table 5.4 shows the results of the CG algorithm using a 10,000 by 10,000 input matrix with 177,782 non-zero elements. The table includes the MOPS obtained for the four test systems and SCMP with five varying clock speeds. The time per iteration is measured in microseconds for all systems excluding SCMP, which is reported in clock cycles due to its lack of hardware. A comparison of SCMP to each test system using its rival's clock speed is

Table 5.4: MOPS Results using Conjugate Gradient.

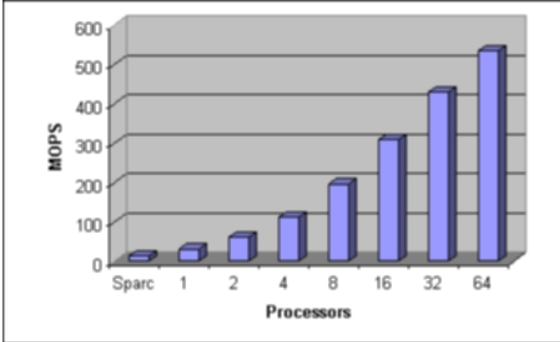
	Time (μs)	MOPS				
G4	42868	19.62				
SPARC	64221	13.01				
Cluster	31352	26.83				
P4	5368.9	156.67				
		MOPS with clock estimate				
	Time (clock cycles)	333MHz	500MHz	700MHz	2.5GHz	5GHz
SCMP-1	13388000	20.92	31.41	43.00	156.07	314.14
SCMP-2	6903400	40.57	60.92	85.29	304.61	609.22
SCMP-4	3724400	75.21	112.92	158.09	564.61	1129.29
SCMP-8	2140700	130.84	196.46	275.05	982.32	1964.64
SCMP-16	1353500	206.94	310.73	435.02	1553.64	3107.28
SCMP-32	974110	287.54	431.75	604.45	2158.74	4317.48
SCMP-64	768780	356.01	534.55	748.36	2672.73	5345.46

shown in Figure 5.3.

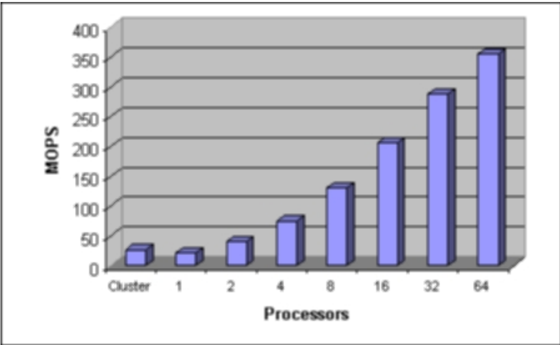
The MOPS metric provided encouraging results. Even when using a highly conservative clock estimate of 333MHz, an eight node SCMP configuration nearly outperformed the Pentium running over seven times faster. The same setup showed that the on-chip network yielded significant performance gains over the interconnected SPARC cluster. Examining the 2.5GHz SCMP clock estimate shows that the two-node simulation doubles the Pentium result, while dominating the other test systems.



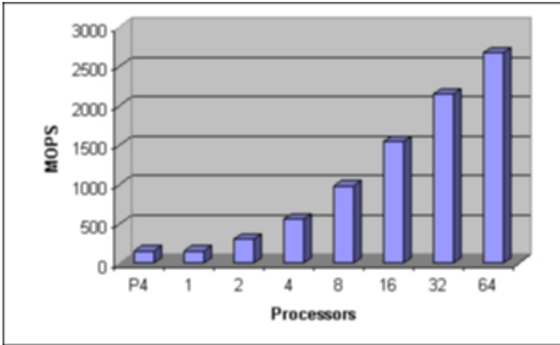
(a) G4 versus SCMP, 700MHz



(b) SPARC versus SCMP, 500MHz

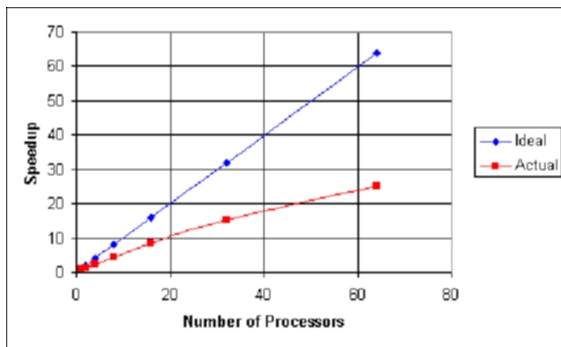


(c) Cluster versus SCMP, 333MHz

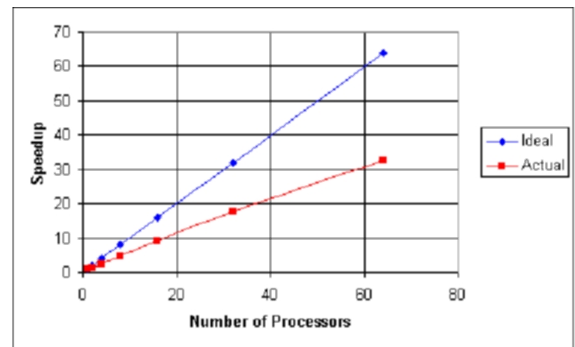


(d) P4 versus SCMP, 2.5GHz

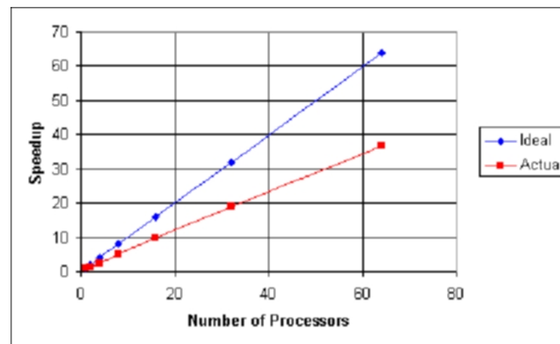
Figure 5.3: CG MOPS Comparison using Rival Clock Speed.



(a) 256 x 256 Matrix



(b) 512 x 512 Matrix



(c) 1024 x 1024 Matrix

Figure 5.4: QR Decomposition Speedup.

5.3 QR Decomposition

5.3.1 Speedup Results

The speedup of the QR decomposition was obtained in a similar manner to the conjugate gradient. The application was simulated on one to sixty-four nodes using three different input matrices. The goal of the metric was to verify that the scatter data layout was effective in providing enough granularity to the larger processor configurations.

Table 5.5: OP Count of QR Algorithm, n x n Matrix, Householder Vector of Length k.

	Operation	Count
	dot = v[1:k-1]^t * v[1:k-1]	2(k-1)
	mu = sqrt(v[0]^2+ dot)	6
	v[0] = -dot/(v[0] + mu)	5
	beta = (2 * v[0]^2) / (dot * v[0]^2)	7
	normalize v	4k
	z^t = v^t * A	2(n-k)^2
	R = A - beta * v * z^t	3(n-k)^2
OPS per iteration		6k + 16 + 5(n-k)^2

Figure 5.4 shows that the QR provided better speedup results than the conjugate gradient. Unlike the CG, the speedup does not tend to level off as more nodes are applied to the problem. Rather, the speedup continues linearly, albeit at a lesser slope than ideal. Furthermore, the figure reveals that as the problem size is increased, the rate of speedup also increases. Bigger matrices were subsequently applied in an attempt to exhibit even greater speedup, but failed due to a lack of memory on the small processor configurations.

5.3.2 MOPS Results

The MOPS count is difficult to calculate for the QR decomposition because each column annihilation manipulates a smaller data set, therefore altering the number of required operations to complete the algorithm. To solve this dilemma, the operations per iteration need to be summed over all Householder vector lengths. The operations per iteration are shown in Table 5.5 while Equation 5.2 provides the corresponding summation and result.

$$TotalOPS_{QR} = \sum_{k=0}^{n} 6k + 16 + 5(n-k)^2 \tag{5.2}$$

$$= \frac{10n^3+3n^2+119n}{6}$$

The QR decomposition was tested with a 1,024 by 1,024 input matrix to calculate the MOPS metric. The metric was beneficial because it revealed a flaw in the original sequential implementation. Initial calculations indicated that a single SCMP node at 500MHz was outperforming each of the test systems, including the P4 at 2.5GHz. Upon further review, it was found that the data layout lead to significant memory hierarchy thrashing [34]. After fixing the problem, the final results in Table 5.6 and Figure 5.5 were attained.

The MOPS results show that SCMP outperformed the G4, Sparc, and cluster systems with four nodes even at the most conservative clock speed estimate. On the other hand, to better the P4, more nodes or a higher clock speed needed to be applied to the problem. Still, at 2.5GHz, the one node configuration is comparable to the P4. This suggests that the on-chip memory of SCMP is more beneficial than the exploited ILP of the test system.

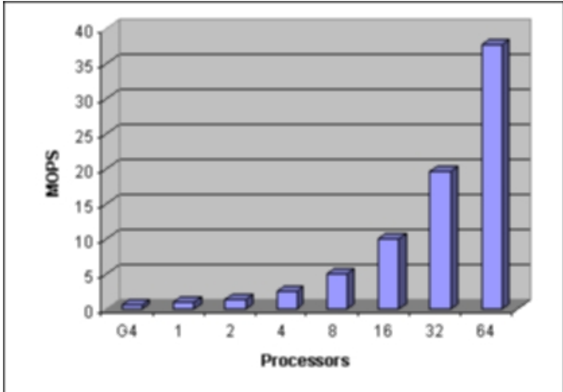
5.4 JPEG

5.4.1 Speedup Results

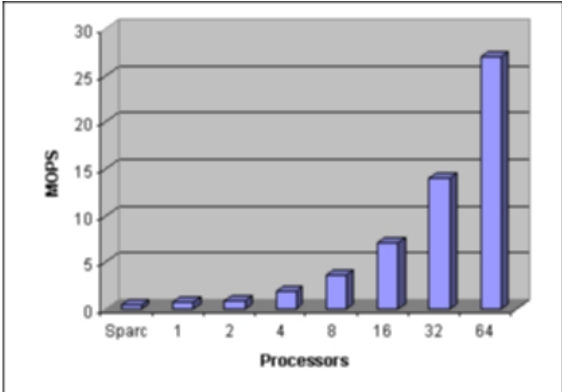
The speedup of the parallel JPEG implementation was again obtained by simulating the algorithm with a variety of SCMP processor configurations. Several images sizes, in both grayscale and color format, were used as input. The results are shown in Figure 5.6 and Figure 5.7, respectively.

Table 5.6: MOPS Results using QR Decomposition.

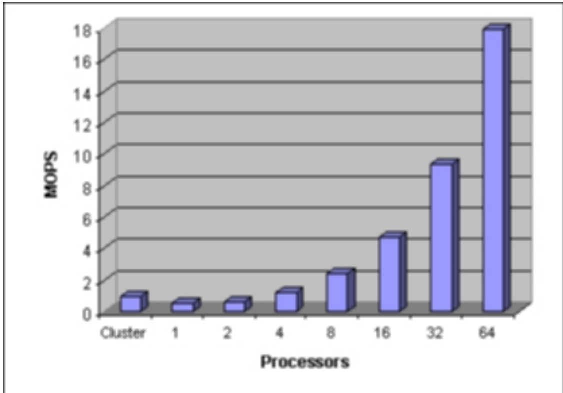
	Time (μs)	MOPS				
G4	47890000	0.58				
SPARC	60240000	0.46				
Cluster	29880000	0.94				
P4	7230000	3.87				
		MOPS with clock estimate				
	Time (clock cycles)	333MHz	500MHz	700MHz	2.5GHz	5GHz
SCMP-1	19054783	0.49	0.73	1.03	3.67	7.35
SCMP-2	15174665	0.61	0.92	1.29	4.61	9.23
SCMP-4	7652354	1.22	1.83	2.56	9.15	18.29
SCMP-8	3858869	2.42	3.63	5.08	18.14	36.28
SCMP-16	1962542	4.75	7.13	9.99	35.67	71.34
SCMP-32	1002271	9.30	13.97	19.56	69.84	139.68
SCMP-64	520200	17.92	26.91	37.68	134.56	269.13



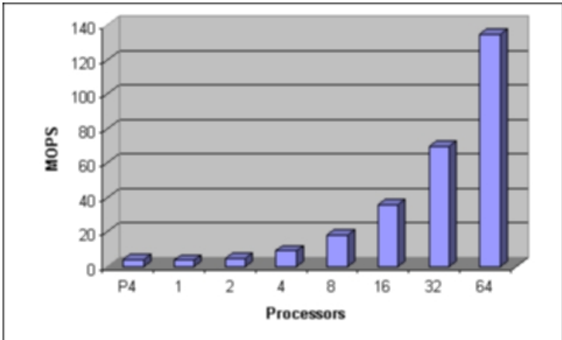
(a) G4 versus SCMP, 700MHz



(b) SPARC versus SCMP, 500MHz

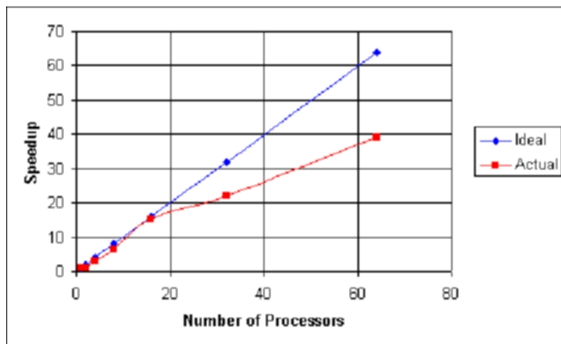


(c) Cluster versus SCMP, 333MHz

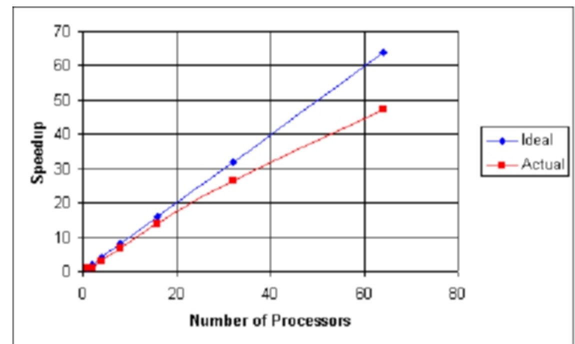


(d) P4 versus SCMP, 2.5GHz

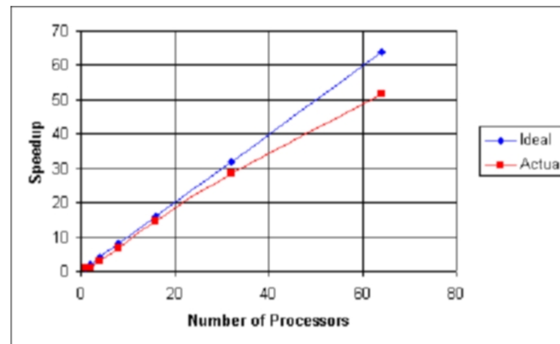
Figure 5.5: QR MOPS Comparison using Rival Clock Speed.



(a) 256 x 256 Grayscale

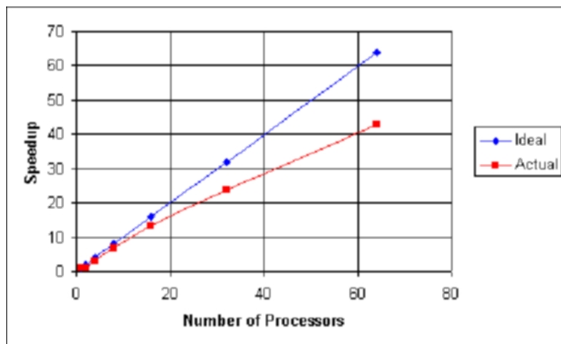


(b) 512 x 512 Grayscale

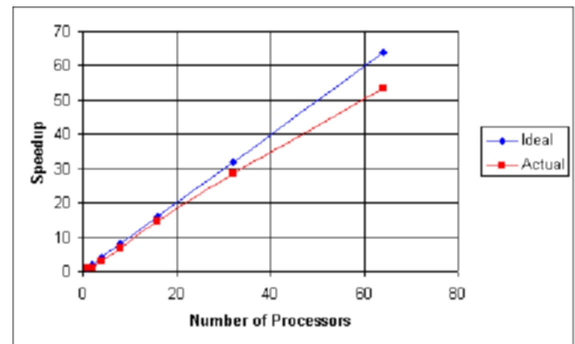


(c) 1024 x 1024 Grayscale

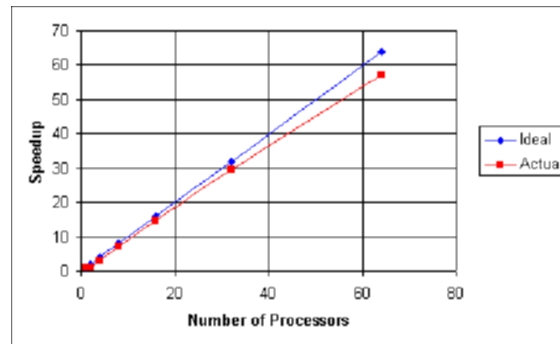
Figure 5.6: JPEG Speedup using Grayscale Images.



(a) 256 x 256 Color



(b) 512 x 512 Color



(c) 1024 x 1024 Color

Figure 5.7: JPEG Speedup using Color Images.

Table 5.7: OP Count of JPEG.

	Operation	Count
	8x8 DCT	640
	8x8 Quantization	64
	DC component encoding	3
	AC component comparisons for run length	63
	AC component encoding (Avg.)	15
Avg. OPS per block		785

The JPEG showed near ideal speedup even as the processor count and image size were increased. Because the three main steps of the algorithm, the DCT, quantization, and Huffman encoding are block independent one expects the measured results. Also, given that node zero is responsible for writing each block and not computation, the slight deviation from ideal speedup is acceptable.

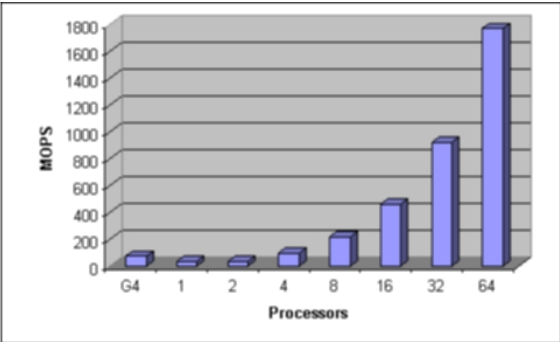
5.4.2 MOPS Results

The MOPS count of the JPEG cannot be directly computed from the algorithm. This is because the OPS required for Huffman encoding vary for each block in the image. In order to quantify the metric, a debug build of the application was constructed to measure the average number of encoded AC components per block. This measurement accounted for both nonzero and zero values (of run-length sixteen) which were shifted into the JPEG bitstream. The test revealed that, on average, each block contained five AC encoded values. Table 5.7 uses this measurement and presents the average number of operations per 8x8 pixel block of the JPEG. When using color images, the operation count is increased to account for the additional red, green, and blue components of each block.

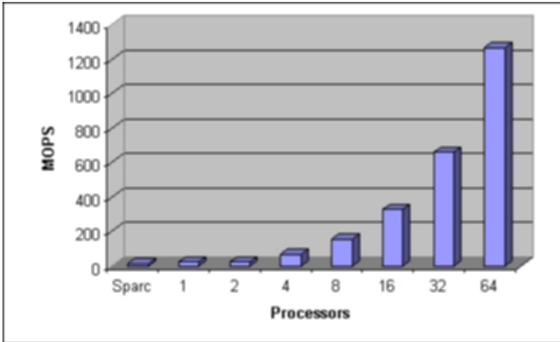
Table 5.8: MOPS Results using JPEG.

	Time (μs)	MOPS				
G4	530000	72.8				
SPARC	2900000	13.3				
Cluster	12340000	3.1				
P4	270000	142.9				
		MOPS with clock estimate				
	Time (clock cycles)	333MHz	500MHz	700MHz	2.5GHz	5GHz
SCMP-1	867683	14.81	22.23	31.13	111.17	222.34
SCMP-2	863153	14.89	22.35	31.29	111.75	223.51
SCMP-4	288432	44.55	66.89	93.64	334.43	668.86
SCMP-8	124453	103.24	155.02	217.02	775.08	1550.16
SCMP-16	58743	218.73	328.42	459.78	1642.08	3284.16
SCMP-32	29244	439.36	659.70	932.57	3298.48	6596.96
SCMP-64	15242	842.97	1265.72	1772.01	6328.62	12657.24

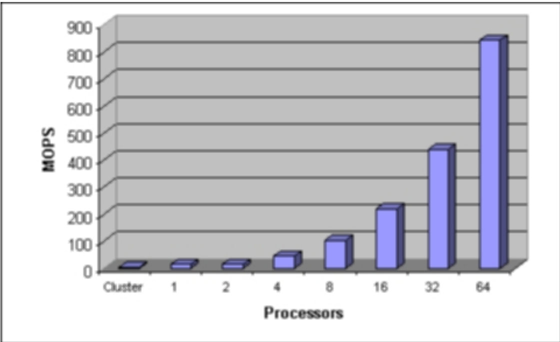
The MOPS attained while encoding a 1024x1024 color image are presented in Table 5.8 and Figure 5.8. The metric revealed promising results for SCMP, especially when using larger processor configurations. The conservative 333MHz clock estimate showed that a sixty-four node setup significantly outperformed the Pentium system. The metric also indicated that the 8x8 block size and large communication costs did not produce an adequate communication to computation ratio for the SPARC cluster. Alternatively, the nominal cost for sending data across a chip allowed SCMP to avoid this pitfall.



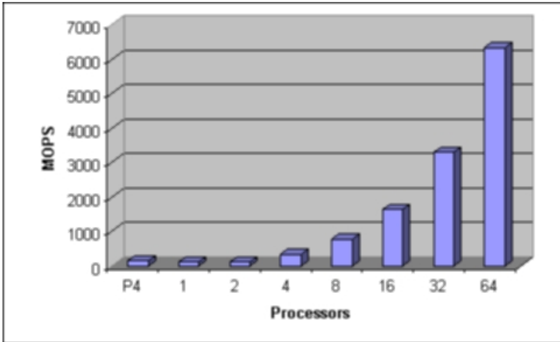
(a) G4 versus SCMP, 700MHz



(b) SPARC versus SCMP, 500MHz



(c) Cluster versus SCMP, 333MHz



(d) P4 versus SCMP, 2.5GHz

Figure 5.8: JPEG MOPS Comparison using Rival Clock Speed.

5.5 Summary

This chapter has compared SCMP to four test systems using the applications that I developed. The results show that the SCMP architecture provides adequate speedup to parallel algorithms which have a reasonable communication to computation ratio. Hence, the performance of the system scales adequately as the number of nodes is increased. Furthermore, the MOPS metric has revealed that, even at a low clock speed, SCMP can outperform current architectures aimed at ILP.

Chapter 6

Conclusions

6.1 Future Work

Although several applications now exist for SCMP, there is still much related research that needs to be accomplished. The following are areas of focus which have been left for future work.

6.1.1 Application Profiling

The SCMP simulator contains several tools for observing the behavior of the network and processor throughout a program's execution, such as the concurrency and message traffic logs. However, these measurements do not aid the application developer a great deal in understanding why his or her parallel program is performing poorly. The reasons why an application does not fulfill expectations can be many. For example, poor data layout, lack of TLP, unexpected network patterns, and process scheduling issues can all contribute an SCMP program's performance problems. To resolve these issues, and possibly others, an

application profiler could be developed to observe where and how clock cycles are consumed. The development of such a tool would require significant changes to both the SCMP simulator and compiler. But, the benefit of the profiler could become invaluable as SCMP research continues.

6.1.2 Source Level Debugging

Currently, the only form of debugging at the SCMP programmer's disposal is the SCMP simulator. The simulator allows one to set a breakpoint based on the clock count or instruction pointer. Regardless, the calculation of either offset from C code is a cumbersome task. It is faster and usually more efficient for the programmer to simply insert numerous print statements to view desired information via standard output. A source level debugger would alleviate the need for constant insertion of temporary code, thereby decreasing development time. The debugger would have to work in conjunction with simulator and probably require extra information to be included by the compiler with the executable.

6.1.3 Parallelizing Compiler

As mentioned in Chapter 2, many single-chip multiprocessors make use of parallelizing compilers. These tools allow sequential applications to be quickly ported to a given parallel architecture. Another benefit of a parallelizing compiler is that, by using standardized benchmarks, such as those from SPEC, comparisons to other systems become significantly easier. But as [7] indicates, the unique architecture of SCMP presents several architectural barriers for the implementation of such a tool. The realization of a parallelizing compiler for SCMP is possible, though. The SUIF toolset already provides the data-flow analysis and partitioning transformations that would be required.

6.2 Summary

This thesis has presented three high-performance applications that I developed for the Single-Chip Message-Passing Parallel Computer. The results show that SCMP provides a significant performance improvement to algorithms which contain adequate amounts of thread-level parallelism. Speedup is maintained as the number of processors is increased, given a sufficient problem size. The results show a speedup of nearly 57 was attained in one of the applications with a sixty-four node configuration. In addition, the MOPS metric indicates that SCMP performs well when compared to other current microprocessors. Over 12.5 GOPS (Giga-OPS) was achieved with the SCMP JPEG implementation.

By using identical cores on a single die connected via a 2-D mesh network, SCMP overcomes the signal propagation and design scalability issues plaguing other current architecture designs. Furthermore, this thesis has shown that the aim to exploit TLP over ILP proves effective in several parallel algorithms. In conclusion, SCMP serves as a viable alternative to the growing barriers computer architects face today.

Bibliography

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*, 1999.

[2] D.W. Apley and Jianjun Shi. The inverse qr decomposition in order recursive calculation of least squares coefficients. In *Proceedings of the American Control Conference*, pages 544–548, 1995.

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

[4] Jonathan Babb, Matthew Frank, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The RAW benchmark suite: Computation structures for general purpose computing. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 134–143, Los Alamitos, CA, 1997. IEEE Computer Society Press.

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

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

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

[8] C. H. Bischof. A parallel qr factorization algorithm using local pivoting. In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 400–499. IEEE Computer Society Press, 1988.

[9] Kenneth Cain, Jose Torres, and Ronald Williams. Rp_stap : Real-time space-time adaptive processing benchmark. Technical report, The MITRE Corporation, 1997.

[10] W.H. Chen, C.H. Smith, and S.C. Fralick. A fast computational algorithm for the discrete cosine transform. *IEEE Transactions on Communications*, 25:1004–1009, 1977.

[11] Lucian Codrescu, D. Scott Wills, and James D. Meindl. Architecture of the atlas chip-multiprocessor: Dynamically parallelizing irregular applications. *IEEE Transactions on Computers*, 50(1):67–82, 2001.

[12] Computational Science Education Project. *Numerical Linear Algebra*, 1995.

[13] G.W. Cook and E.J Delp. The use of high performance computing in jpeg image compression. In *1993 Conference Record of the Twenty-Seventh Asilomar Conference on Signals, Systems and Computers*, pages 846–851, 1993.

[14] M. Cosnard and Y. Robert. Complexity of parallel qr factorization. *J. ACM*, 33(4):712–723, 1986.

[15] David E. Culler, Jaswinder P. Singh, and Anoop Gupta. *Parallel Computer Architecture.* Morgan Kaufmann Publishers, 1999.

[16] S. Emami and S.I. Miller. Signal selection for non-symmetric sources in two dimensions. In *Proceedings of the IEEE Southeast Conference, 1992*, pages 461–464, 1992.

[17] Krisztian Flautner, Richard Uhlig, Steven K. Reinhardt, and Trevor N. Mudge. Thread level parallelism and interactive performance of desktop applications. In *Architectural Support for Programming Languages and Operating Systems*, pages 129–138, 2000.

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

[19] Gene H. Golub and Charles Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1996.

[20] Rajiv Gupta, Michael Epstein, and Michael Whelan. The design of a risc based multiprocessor chip. In *Proceedings of the 1990 conference on Supercomputing*, pages 920–929. IEEE Computer Society Press, 1990.

[21] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[22] Lance Hammond and K. Olukotun. Considerations in the design of hydra: a multiprocessor-on-a-chip microarchitecture. Technical report, Stanford University, 1998.

[23] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69. ACM Press, 1998.

[24] M. Harteneck and R.W. Stewart. Acoustic echo cancellation using a pseudo-linear regression and qr-decomposition. In *IEEE International Symposium on Circuits and Systems*, pages 233–236, 1996.

[25] Kai Hwang and Zhiwei Xu. *Scalable Parallel Computing: Technology Architecture, and Programming*. McGraw-Hill, 1998.

[26] International Standard Organization. *Digital Compression and Coding of Continuous-Tone Still Images*, t.81 edition, September 1992.

[27] T. Kodaka, K. Kimura, and H. Kasahara. Multigrain parallel processing for jpeg encoding on a single chip multiprocessor. In *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 57–63, 2002.

[28] M. Kovac, N. Ranganathan, and M. Zagar. A prototype vlsi chip architecture for jpeg image compression. In *European Design and Test Conference*, pages 2–6, 1995.

[29] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

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

[31] Satoshi Matsushita. Design experience of a chip multiprocessor merlot and expectation to functional verification. In *Proceedings of the 15th international symposium on System Synthesis*, pages 103–108. ACM Press, 2002.

[32] M.K. Ng. Nonlinear image restoration using fft-based conjugate gradient methods. In *Proceedings of the IEEE International Conference on Image Processing*, pages 41–44, 1995.

[33] J.S. Patrick, J.L Sanders, L.S. DeBrunner, V.E. DeBrunner, and S. Radharkrishnan. Jpeg compression/decompression via parallel processing. In *1996 Conference Record of the Thirtieth Asilomar Conference on Signals, Systems and Computers*, pages 596–600, 1996.

[34] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[35] S.V. Ramaswamy and G.D Miller. Multiprocessor dsp architectures that implement the fct based jpeg still picture image compression algorithm with arithmetic encoding. *IEEE Transactions on Consumer Electronics*, 39:1–5, 1993.

[36] L. Sang-Won, S. Yun-Seob, K. Soo-Won, O. Hyeong-Cheol, and H. Woo-Jang. Raptor: A single chip multiprocessor. In *The First IEEE Asia Pacific Conference on ASICs*, pages 217–220, 1999.

[37] Karthikeyan Sankaralingam, Stephen W. Keckler, William R. Mark, and Doug Burger. Universal mechanisms for data-parallel architectures. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 303. IEEE Computer Society, 2003.

[38] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433. ACM Press, 2003.

[39] Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French. A performance analysis of pim, stream processing, and tiled processing on memory-intensive signal processing kernels. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 410–421. ACM Press, 2003.

[40] Michael Bedford Taylor. The raw processor specification. Technical report, Massachusetts Institute of Technology, 2003.

[41] Titan Systems Corporation. *Specifications for the Stressmarks of the Data Intensive Suite Benchmark Project*, 2000.

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

[43] G.K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34:31–44, 1992.

[44] A.R. Weiss. The standardization of embedded benchmarking: pitfalls and opportunities. In *International Conference on Computer Design*, pages 492–508, 1999.

[45] 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.

[46] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[47] B.B Zhou and R.P. Brent. Parallel implementation of qrd algorithms on the fujitsu ap1000. Technical report, The Australian National University, 1993.

Vita

William Wesley Dickenson was born on September 1, 1979 in Big Stone Gap, Virginia. In 1998, he graduated from Powell Valley High School with honors. Following high school, he enrolled at Virginia Polytechnic Institute and State University in Computer Engineering. He completed his undergraduate studies with a Bachelor of Science in 2003, graduating Magna Cum Laude.

Wesley will complete his Master of Science degree in Computer Engineering in May 2004. A Graduate Teaching Assistantship supported his studies. Upon graduation, he will begin work with Sparta, a federal government contractor, in Centreville, Virginia.