Application Benchmarks for the SCMP: Single-Chip Message-Passing Computer

Jignesh Shah

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

August 25, 2003

Bradley Department of Electrical and Computer Engineering

Blacksburg, Virginia

Keywords: SCMP, Message Passing, Benchmarks, Parallel Computer

Application Benchmarks for the SCMP: Single-Chip Message-Passing Computer

Jignesh Shah

Abstract

As transistor feature sizes continue to shrink, it will become feasible, and for a number of reasons more efficient, to include multiple processors on a single chip. The SCMP system being developed at Virginia Tech includes up to 64 processors on a chip, connected in a 2-D mesh. On-chip memory is included with each processor, and the architecture includes support for communication and the execution of parallel threads. As with any new computer architecture, benchmark kernels and applications are needed to guide the design and development, as well as to quantify the system performance. This thesis presents several benchmarks that have been developed for or ported to SCMP. Discussion of the benchmark algorithms and their implementations is included, as well as an analysis of the system performance. The thesis also includes discussion of the programming environment available for developing parallel applications for SCMP.

Acknowledgements

While working on my research thesis, a lot of people have helped and motivated me. I would like to thank all of them for their continued support.

Firstly, I would like to thank Dr. Mac Baker, my primary advisor, for spending the time with me during this thesis. He was very approachable anytime of the day, and was ready to answer any questions I had asked.

Secondly, I would also like to thank Dr. Reinhard Laubenbacher at Virginia Bioinformatics Institute for supporting me for my Masters degree and being a great mentor. I really enjoyed working with him and learned a great deal too.

I would also like to thank Dr. Peter Athanas and Dr. William Baumann for their support and their willingness to serve on my committee.

The people at the SCMP lab have been very friendly and helpful and it was great working with them. Lastly, I would like to thank all my friends who were always there for me whenever I needed them.

Jignesh Shah

August 2003

To my parents, brother, sister and friends, without whom nothing would have been possible.

Index

List of Figures

1. Introduction

1.1 Introduction

The success of advanced computational and mathematical sciences has fueled the need for small, fast and cheap computing power. We have been able to fulfill these requirements till now, and will also be able to for sometime in the near future. But at some point, it will become very difficult to satisfy these challenges. One way these challenges could by fulfilled would be by building and designing parallel computers. The goal of parallel computing is to improve performance along with providing the same results as a single processor computer does, but by using multiple processors. Most of the time, with increase in the number of processors, the speed of execution of an application also increases.

Currently, the problem of increasing computational power is being tackled by increasing the clock speed of processors and also introducing and increasing instruction level parallelism (ILP) in applications [Hennessy02]. But, in the future clock speeds will be limited by the effects of on-chip wire latencies. Improvement in technology has greatly reduced the transistor size, thus reducing the diameter of interconnect wires to match the transistor size. As the wire diameter shrinks, the resistance of the wire increases, leading to increased latencies when transmitting signals. These longer latencies have an effect on clock speed of processors and limiting their performance. To overcome this hurdle, computers should be designed that have shorter interconnect wires.

The amount of ILP that can be extracted from an application is also reaching a point of diminishing returns. The cost of extracting the ILP is not proportional to the increase in performance. To continue the increase in performance, architectures should be designed that not only exploit ILP but also introduce thread level parallelism (TLP).

A promising solution to overcome the problem of increased latencies and decreased returns of implementing ILP can be solved by developing single-chip parallel computers. These systems would include a number of small, simple, low-power consuming processors with every processor having its own on-chip memory. Such a computer would have low latencies due to shorter interconnect wires and also the presence of multiple processors would allow applications to take advantage of TLP.

One such system under development is the Single-chip Message-passing (SCMP) computer [Baker02], currently being researched at Virginia Tech. The idea behind the computer is that all the processors will be small enough to fit on a single chip, and every processor will have its own on-chip memory. Through this architecture, it is possible to introduce thread level parallelism, i.e., a thread is allowed to execute on a single processor. Execution of multiple threads on multiple processors will lead to the faster execution of a program and thus reducing the time required for computation. This system includes up to 64 processors on a single chip, connected in a 2-D mesh.

When research on a new system begins, all the other work related to it also starts along with the development of the system. During the design phase of a system, simulations are be required to give an approximation of the gain in performance and speedup the new system might provide. To come up with these numbers, numerous benchmarks and applications need to be designed, written and tested on the machine. These benchmarks should be industry standard benchmarks, if possible, which many of the present parallel systems utilize. Using the industry standard benchmarks will help compare the performance of the new system with the ones already existing.

This thesis focuses on taking standard benchmarks and applications and porting them to the SCMP system, and then comparing its performance with other parallel systems. All the benchmarks and applications developed were written in the C language and compiled using the SCMP C compiler. The programs were then run on the SCMP system simulator. The results were then compared with the single processor execution of the program and its parallel implementation.

1.2 Thesis Organization

Before getting into the implementation details of the algorithms it is required to get a good understanding about the SCMP architecture. Chapter 2 discusses the motivation for the thesis and the need for benchmarks for parallel computers. Chapter 3 describes the general architecture of the SCMP computer. Chapter 4 focuses on the tools that are available for developing application and benchmarks for the SCMP. Chapter 5 details all the applications that have been developed for the SCMP and also the general algorithmic approach needed to develop applications. Chapter 6 analyzes the output and results of all the experiments. Chapter 7 is a conclusion about the work done and what could be done in the future.

2. Background and Related Research

A parallel computer is a "collection of processing elements that communicate and cooperate to solve large problems fast" [Almasi89]. Parallel computers are broadly divided into two categories, shared address space computers and message passing computers.

A shared address space multiprocessor system is one in which communication occurs implicitly using conventional memory access schemes. These shared memory multiple-instruction multiple-data (MIMD) architectures may have independent L1 caches but share an L2 cache. Memory latency in shared memory systems could be considerable since global memory accesses must travel across the system so that all of the processors can access each memory location. Examples of some of the shared address space multiprocessors currently under use are the SGI Origin, Power4, Hydra and Blue Gene.

In a message passing multi-computer system, any type of communication between the processor, memory and I/O is by sending explicit messages from one processor to another. Message passing systems are harder to program than shared memory systems. This is because every communication step or accessing a memory of another processor has to take place by sending explicit messages and then receiving a response from the processor.

2.1 Related Research

Early message-passing systems included the Cosmic Cube [Seitz85], the Mosaic C system [Athas88], and the MIT J- and M-Machine [Noakes93, Fillo95]. These systems showed that through fine-grained message passing, it was possible to build a large scale, multi-computer message-passing system. The Pica system [Wills97] developed at Georgia Tech had an active message style with integrated processor, network components and a small size local memory. The Raw [Agarwal97] processor is another message-passing architecture with multiple processors on a single chip. The Raw processor exploits Thread Level Parallelism (TLP) and also Instruction Level Parallelism (ILP) of applications. However, it includes a very small amount of memory (32 KB data cache, 32 KB instruction cache) is included on-chip with the processor. Simultaneous multithreading, or SMT [Krishnan99] presents an alternative processor design for exploiting thread level parallelism. SMT processors have a large number of functional units, with every thread competing for the control of the function unit. A thread in the SMT may only include modest amounts of ILP and would not be able to fully utilize all of the functional units of the processor. Other threads may use the remaining functional units, which would in turn lead to an overall increase in performance, utilization and throughput. The SMT design does not address the wire latency issues that were discussed earlier, and also the support for a large number of threads would be a limiting factor.

The SCMP system falls in the second category, being a message passing multi computer system. Every processor has its own memory associated with it, separate from the other processors. There is no shared location that is accessible to all the processors. A memory access from another processor has to take place through explicit message passing from one processor to another. Some of the important features of the SCMP include high-bandwidth and low-latency communication. A detailed description of the SCMP architecture is given in the following chapter.

2.2 Parallel Computer Benchmarks

Rapidly changing technology has resulted in the development of various different kinds of parallel computers. Parallel program execution schemes have emerged as a general, widely used computer systems technology, which is no longer reserved for just super computers. Today, off the shelf computers and servers include multiple processors capable of delivering high performance. The presence and wide availability of parallel computer systems necessitates development for adequate benchmarks that measure the performance in a fair manner. Currently, there exists benchmarks that are used for shared memory parallel computers, but there are only a few that measure the performance of a message-passing multi-computer system like the SCMP. Several parallel computer benchmark suites exist today, some of which are discussed below.

- **SPLASH-2** – Stanford Parallel Application for Shared Memory-2 (Splash-2) [Woo95] is a suite of parallel applications that was developed to facilitate the study of centralized and distributed shared-address-space multiprocessors. These benchmarks study the computational load balance, communication to computation ratio and traffic needs, important working set sizes, and issues related to spatial locality, as well as how these properties scale with problem size and the number of processors. These benchmarks were also developed to compare the architecture of other shared-memory multiprocessors amongst each other.
- **CoMet** – ComMet [Ganapati93] , developed by Oregon Graduate Institute of Science & Technology, is a benchmark suite developed for message-passing

parallel computers mainly consisting of matrix related kernels written in C. Its kernels measure the machine's basic communication characteristics under light and heavy load.

- **Perfect Benchmarks** – Perfect benchmarks [Berry89] were used to measure high-performance computer systems in the early 90s. They included several standard sequential programs that were converted into parallel programs, and executed on shared memory systems.
- **SPEComp** – SPEComp [Ashlot01] benchmarks was developed by several industry leaders such as Intel Corp and Sun Microsystems to evaluate the performance of mid-size parallel servers. It was designed to run with relative ease and moderate use of resources. These benchmarks addressed scientific, industrial and customer benchmarking needs.
- **Linpack** – The Linpack (Liner Algebra) [Dongarra79] benchmark suite was developed by Jack Dongarra, and is widely used to measure performances of both shared-memory and message-passing architecture computers. Linpack is a package of linear algebra routines that are widely used in the operating system kernel. Linpack was originally developed as a collection of Fortran subroutines that analyzed and solved linear equations in the late 70's and early 80's. Currently Linpack has been largely superceded by Linear Algebra Package (Lapack), which has been designed to run efficiently on shared memory and vector super computers.
- **TPC Benchmarks** – The Transaction Processing Performance Council (TPC) [Gray93] benchmarks is a collection of transaction processing and database

benchmarks. TPC-C is the industry standard for measuring the performance and scalability of OLTP systems. It includes a broad cross section of database functionality including update, queued transactions and database inquiry.

All the benchmarks discussed above have been designed for different computer architectures and measure different characteristics. Any single benchmark suite cannot accurately measure the performance of a parallel machine. This is due to the fact that every computer has its own set of instructions and is developed for a specific purpose. Evaluating the performance of a parallel computer is a function of various issues such as the architecture, memory management, I/O schemes and its instruction set. Thus the benchmarks for one particular kind of parallel computer might be inappropriate for another computer. The applications developed should also be written to exploit the new features of the system. Existing benchmarks fall short on these counts when used to evaluate distributed-memory message-passing computers. A benchmark for a shared memory parallel computer, might not give accurate performance measures for a message passing parallel system. This is also true for the SCMP computer. Existing parallel computer benchmarks do not run directly on the SCMP computer. Instead they have to be converted to a language that could be understood and interpreted by the SCMP processors. This requires taking existing industry standard benchmarks and converting them to benchmarks that is understood by the SCMP compiler.

For this thesis, a few of the application benchmarks developed have been converted from the shared-memory SPLASH-2 benchmark suite. Since a message-passing version of the SPLASH-2 does not exist, the algorithms were closely studied and converted to a message-passing algorithm performing the same functionality.

Benchmarks are generally divided into kernel and application benchmarks. Kernel benchmarks attempt to span a wide range of applications and tasks that are performed by the operating system, and include the most frequently encountered computationally intensive kinds of problems. These are low-level benchmarking programs that are designed to measure the basic architectural implementation of the parallel computer. Kernel benchmarks may also consist of larger code segments extracted from real applications. Example kernel benchmarks include Livermore, NAS kernels, LINPACK benchmarks.

Application benchmarks are implementation of a full application, which may include the use of some kernel benchmark programs. Well-know application benchmarks include the SPEC, Perfect and Eurobean benchmark suites. These benchmarks accurately reflect the performance characteristics of a machine with respect to the specific classes of real world applications. Two of the SPLASH-2 kernel benchmarks, Integer Radix Sort and LU Factorization, have been implemented for the SCMP computer. Radix sort is an integer sorting algorithm and LU Factorization is a method used to find the solution for a set of equations. Other applications such as the Smith-Waterman sequence alignment, the Median filter and the Sobel's edge detection were also implemented. Smith-Waterman's sequence alignment is widely used in bioinformatics, which requires a huge database to do the comparison. Median filter is an image-processing algorithm used to correct an image with salt and pepper noise. Sobel's edge detection algorithm is image-processing algorithm that operates on every pixel of the image.

3. SCMP Architecture

The SCMP (Single-chip message-passing) system is a single-chip parallel computer with a tiled architecture with the processors arranged in a two-dimensional mesh as shown in Figure 3.1. The SCMP parallel computer is designed to exploit thread-level parallelism using localized, structured wiring to reduce design and testing costs and improve power consumption [Baker02]. Each of the SCMP tiles consists of a CPU, memory and networking components.

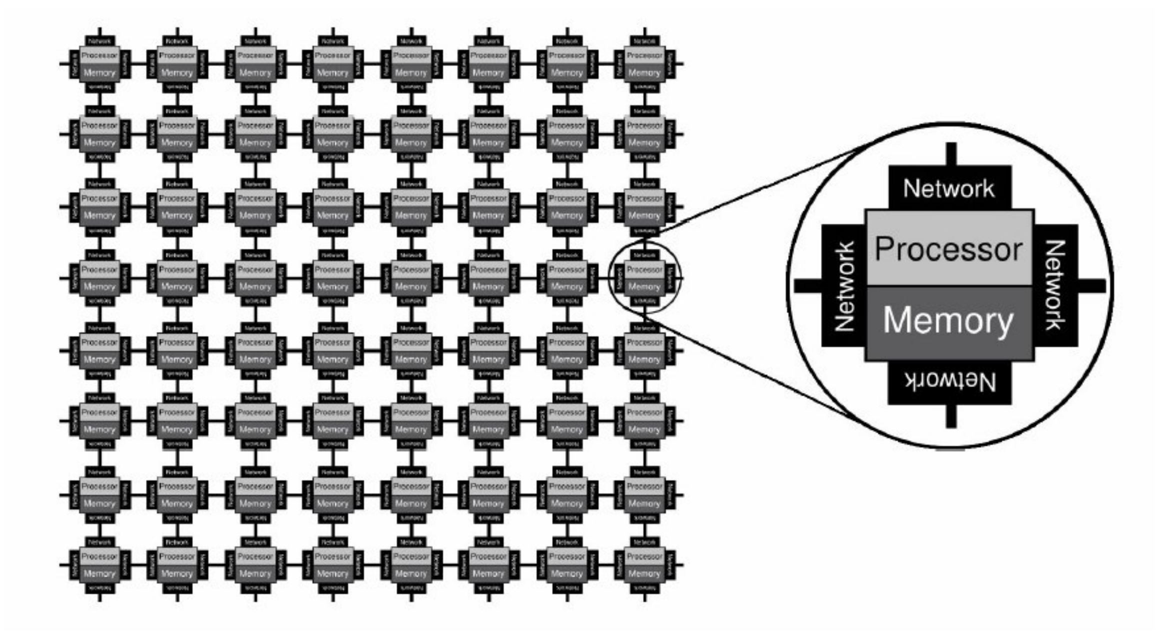


Figure 3.1: *SCMP system with 64 tiles. Each tile contains a CPU, memory and a network interface.*

3.1 CPU Architecture

Each of the SCMP CPUs is a 32-bit RISC core with custom features enabling message passing and multiple threads of execution. The CPU is controlled through a 4-

stage pipeline. A SCMP processor has up to 8MB of local memory associated with it and also includes hardware support for executing up to 16 threads. A detailed diagram of the SCMP node is shown in Figure 3.2

An individual processor of the SCMP grid may not be as powerful as the state-of-the-art single processor system, but having up to 64 processors working in parallel, each operating at a high clock speed will result in substantial speedup. One of the major factors for the speedup is the thread-level parallelism (TLP) feature of the SCMP system. The high-bandwidth and low latency communication also helps in exploiting an application's TLP.

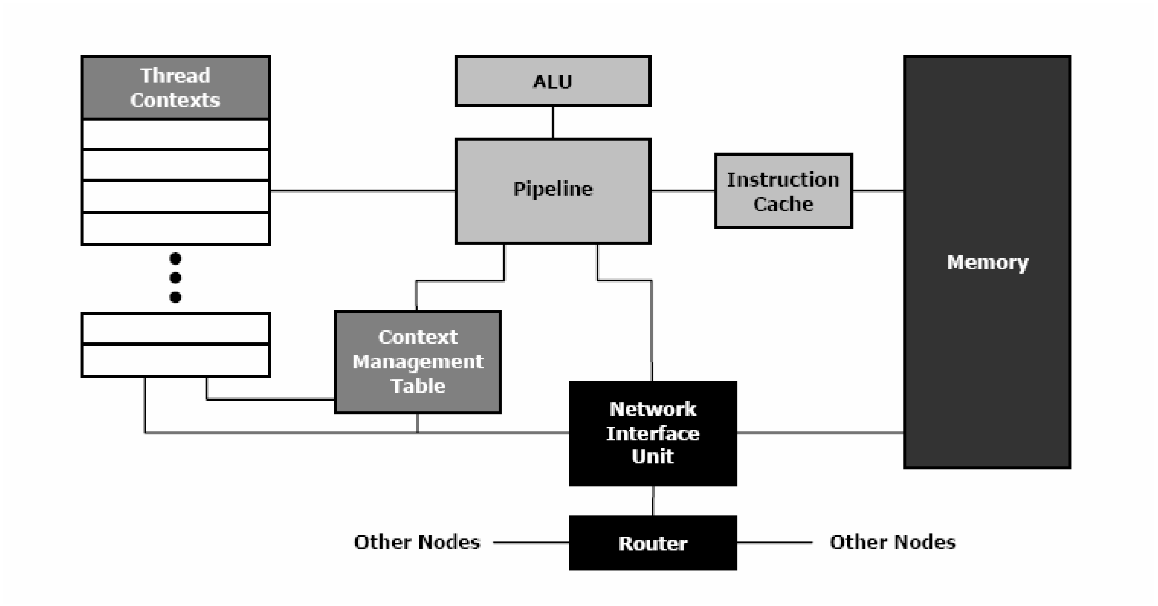


Figure 3.2: *Detailed diagram of a single SCMP node.*

Each processor includes hardware support for up to 16 threads, with non-preemptive round-robin scheduling. Each thread on a SCMP processor has its own associated context, which includes a block of 32 registers. A Special register called the Active Thread Register (ATR) indicates which thread is currently executing on a processor. To switch threads, a new value is written into the ATR and the pipeline is flushed. Since each thread has its own block of registers, it is not necessary to save and restore registers to memory when switching threads.

Each processor's threads are created, managed and scheduled using the Context Management Table (CMT), which has 16 entries, one for each thread context. Each entry contains the state information for a thread. The registers are organized in a 2-D structure of 16 contexts with 32 registers each. The CMT, along with the ATR and the Instruction Pointer (IP) special registers, is used to schedule threads.

3.2 Message Passing Mechanism

The network interface in the SCMP is designed in a manner such that each processor has connections to its four nearest neighbors (North, South, East and West) and dimension-order wormhole routing is used to communicate with other processors in the system. Wormhole routing is primarily used in multiprocessor systems for message routing. When a message arrives at a CPU for forwarding, the CPU examines the header of the message and directs it to the next CPU which will then be directed to the destination CPU. This kind of routing reduces delay since the CPU does not wait for the whole message to arrive and resulting in faster message communication speeds. The SCMP system supports thread and data messages. Thread messages are used to create a new thread on a different processor, whereas data messages transfer data from the local memory of one processor to another processor. Every thread message contains an instruction pointer (IP) and also the context register values. The CPU on which the thread is created handles the new thread data by allocating an entry in the context management table. When sending a data message, the data can be injected into the network from a thread's registers or directly from memory. Similarly, when a message is received, the message data is placed directly into a thread's registers or memory. Data messages consists of a base memory address, an address stride and the number of data words that need to be sent. A buffer is not required while sending or receiving messages.

Since messages in the SCMP network are dynamic, with no prior knowledge or statistics, dimension order routing is used to route the messages. Dimension order routing, which is the selection of successive channels to follow a specific order based on the dimensions of a multidimensional network, is used to route the messages. Thus, every message in the SCMP is first routed in the X dimension and then along the Y dimension.

3.3 On-chip Memory

Each CPU of the SCMP system has up to 8MB of local memory associated with it. Every processor has direct access to its local memory, and not to any other processor's local memory. Since the memory size is comparatively smaller than current computers, it allows memory access in one or two clock cycles, thus increasing the average data access speed. To access the memory of another processor, messages have to be explicitly sent to that processor for the information required.

The instruction cache, the pipeline and the Network Interface Unit (NIU) have access to a processor's local memory. To coordinate these accesses, a custom-designed memory controller is also used.

3.4 Network Architecture

The SCMP network and processor-network interface are designed to minimize overhead and reduce communication latency due to message passing. Each node consists of a network interface unit (NIU) and a router. The NIU is responsible for collecting data from a local node to be sent across the network, as well as receiving messages from the network. Messages are sent from one processor to another without any additional handshaking required. Since there is no hardware restriction on the messages being sent, the programmer has to add synchronization logic in the software.

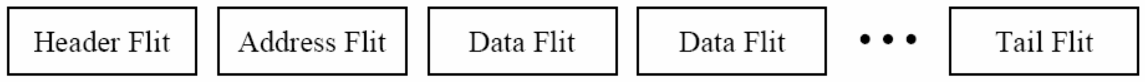


Figure 3.3a: *A complete message in the SCMP network*

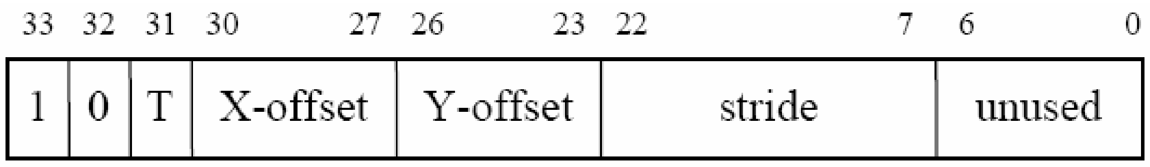


Figure 3.3b: *Header flit. The 'T' bit in field 31 indicates message type, '1' for thread and '0' for data.*

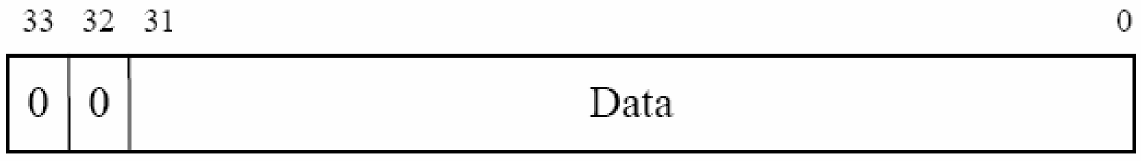


Figure 3.3c: *Data flit. Bit 32 will be set to '1' to indicate a tail flit.*

Messages in the SCMP network are broken into flow-control digits, or flits, representing individual pieces of data that are atomically transmitted through the network [Baker03]. Each message contains a header flit, an address flit, zero or more data flits, and a tail flit. The header flit indicates the message type, i.e. either a thread or a data message, destination node information. If it is a data message, the header flit also contains information about the stride length between memory addresses of consecutive values. Figure 3.3 illustrates the message format used for the SCMP.

A message-passing configuration was chosen for the SCMP to simplify the hardware design and reduce the on-chip interconnect wire lengths. Since the 2-D mesh network requires that each node be connected to only its four nearest neighbors, the wire lengths remain short, allowing high-bandwidth, low-latency network connections. The routers communicate with each other using asynchronous handshaking protocol, which eliminates the need for a synchronized global clock signal and associated clock skew problems.

4. Programming for SCMP

The SCMP has a C compiler, which makes it easier for programmers to develop applications for it. An instruction set simulator has also been developed which executes the application developed and written in C. The simulator collects performance data and other metering information. The following sections give an overview of the compiler and the simulator. The subsequent sections introduce the message-passing paradigm for parallel application development on the SCMP.

4.1 SUIF – C Compiler

To support the development of applications and benchmarks, the SUIF compiler toolset has been ported to the SCMP system, and an assembler and linker have been developed. The compiler follows the standard C program structure for application writing. Libraries of subroutines for communication and parallelism that can be called from C programs have been implemented for the SCMP. These libraries provide low-level functions for sending data between processors, for creating threads on other processors and also include barrier and synchronization functions. The standard C libraries that have been implemented for the SCMP architecture are the:

- stdio.h – Standard library used for Input/Output with streams
- string.h – Library used to manipulate C strings
- math.h – Library for mathematic operations including support for floating point data
- stdlib.h – General purpose standard C library, including memory allocation, process control, conversions and other functions.

4.2 SCMP specific C Libraries

SCMP has its own set of new and different features that the other traditional computer systems do not have. There are no standard libraries available that can be ported to support these features. To include support for these features, along with support for data and thread messages, a set of libraries has been developed, which include files such as scmp.h, thread.h and broadcast.h etc.

4.2.1 scmp.h

The scmp.h files contains processor specific functions that an application programmer might need to know. This library can be further divided into the following categories:

4.2.1.1 SCMP processor specific functions

The set of functions in this library helps the application programmer find out specific details about the processor. It includes functions to find what is the current node of execution, how many nodes are activated in the whole system etc. Some of the SCMP specific system calls are outlined below:

- getNir() – This function returns the current node of execution in the program.
- getXDim() – Returns the number of processors in the X dimension from the active SCMP processors grid. The number of active processors is specified in the simulator configuration file.

- getYDim() – Returns the number of processors in the Y dimension from the active SCMP processors grid.

Node 0 of the SCMP grid is responsible for activating the processors before the execution of a program. This is done by Node 0 sending messages to all the other processors in the gird. The sequence of this initialization of processors in the SCMP grid is shown in the Figure 4.1. Node 0 first sends messages to all the nodes in the first row, and then all the activated nodes in the first row activate all the processors in its column. This reduces the congestion and latency in the SCMP network. The broadcasting mechanism is illustrated in the following figure:

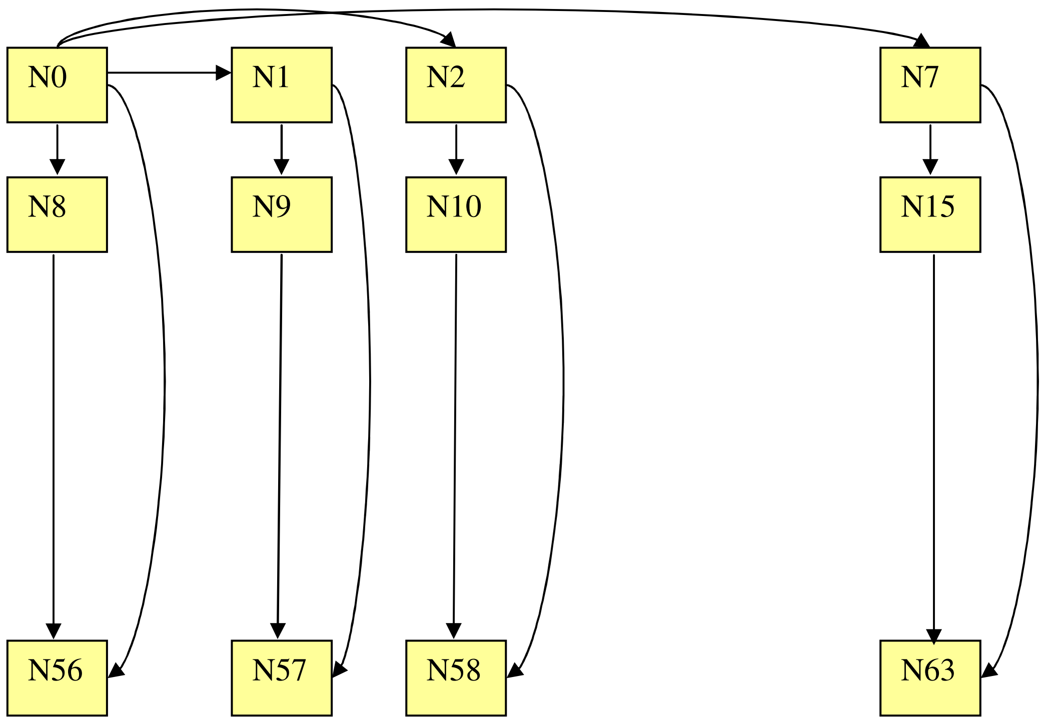


Figure 4.1: *Processor initialization sequence during the start of a program*

4.2.1.2 Communication specific functions

Since SCMP is a message passing system, one of the most important features a programmer might need is to send and receive messages from a different processor on the chip. These send functions are defined in the library file send.h. The file contains the following functions:

- *SendDataBlock* – Send a continuous block of data from one processor to another
- *SendDataIntValue* – Send a single value of type integer from one processor to another.
- *SendDataFloatValue* – Send a single value of type float from one processor to another.

4.2.1.3 Thread handling functions

Thread handling functions are required to create threads on a processor. To create a thread on a specific processor, it is also necessary to specify what function the thread should execute. The createThread function takes in parameters such as the destination processor where the thread is supposed to be created, the function name, callback function name and the number of arguments and the arguments. createThread has the following syntax:

```
void createThread( int dst_node, void (*addr)(), void (*callback_fn)(), ... );
```

A callback function is a function that is executed upon successful completion of the thread. A common use of the *createThread* function in the SCMP is when a processor needs some data from another processor. To achieve this task, the requesting processor creates the thread on the processor that has the data, and passes the function calls such as *SendDataBlock* with its own processor as the destination parameter. After the thread is created and executed by the processor that has the data, it sends it using the *SendDataBlock* function.

Active threads are suspended using the *suspendThread* function in the program. The function suspends the execution of the current thread, and switches the context to the next thread. Whenever the control comes back to the thread, then all the saved information is loaded back to the processor. A thread could be suspended if the processor is waiting for a synchronization variable, and during this time, it could do some other useful work. An example of a common use of the *suspendThread* function is shown in the following pseudo code:

```
Code…
while( synchronization variable not set)
    suspendThread();
code….
```

Example showing the use of the *suspendThread* function

4.3 Simulator

During the development stage of any new processor, it is necessary to have a simulator which will predict the behavior of the system when programs are executed in it.

Scmpsim is the simulator for the SCMP system. It includes both instruction-level simulation of each processor and flit-level simulation of the interconnection network, allowing for the accurate modeling of the system performance. The simulator is Linux based and includes both an XWindows interface as well as the regular command line interface. The XWindow interface shows the current state of all the nodes in the system. A screen shot of the simulator is shown in Figure 4.2. A green square indicates that the node is currently executing an instruction, a red node indicates a stalled processor and a black node represents an inactive processor.

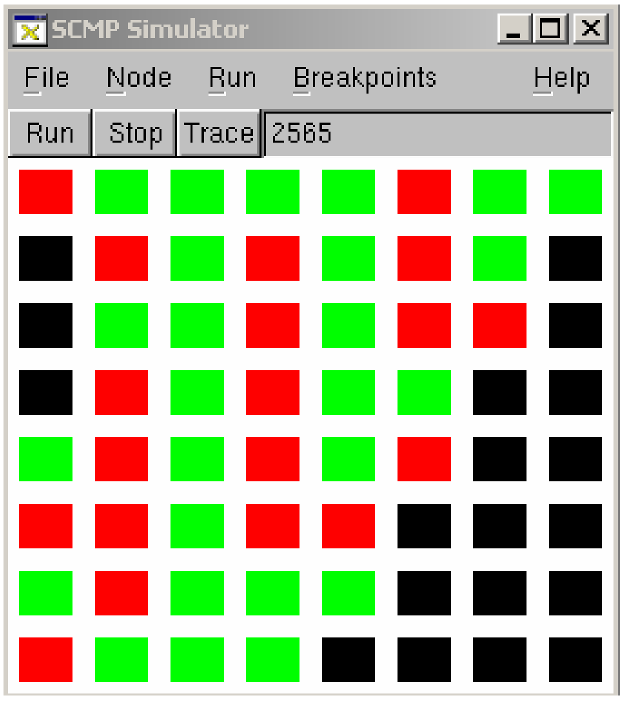
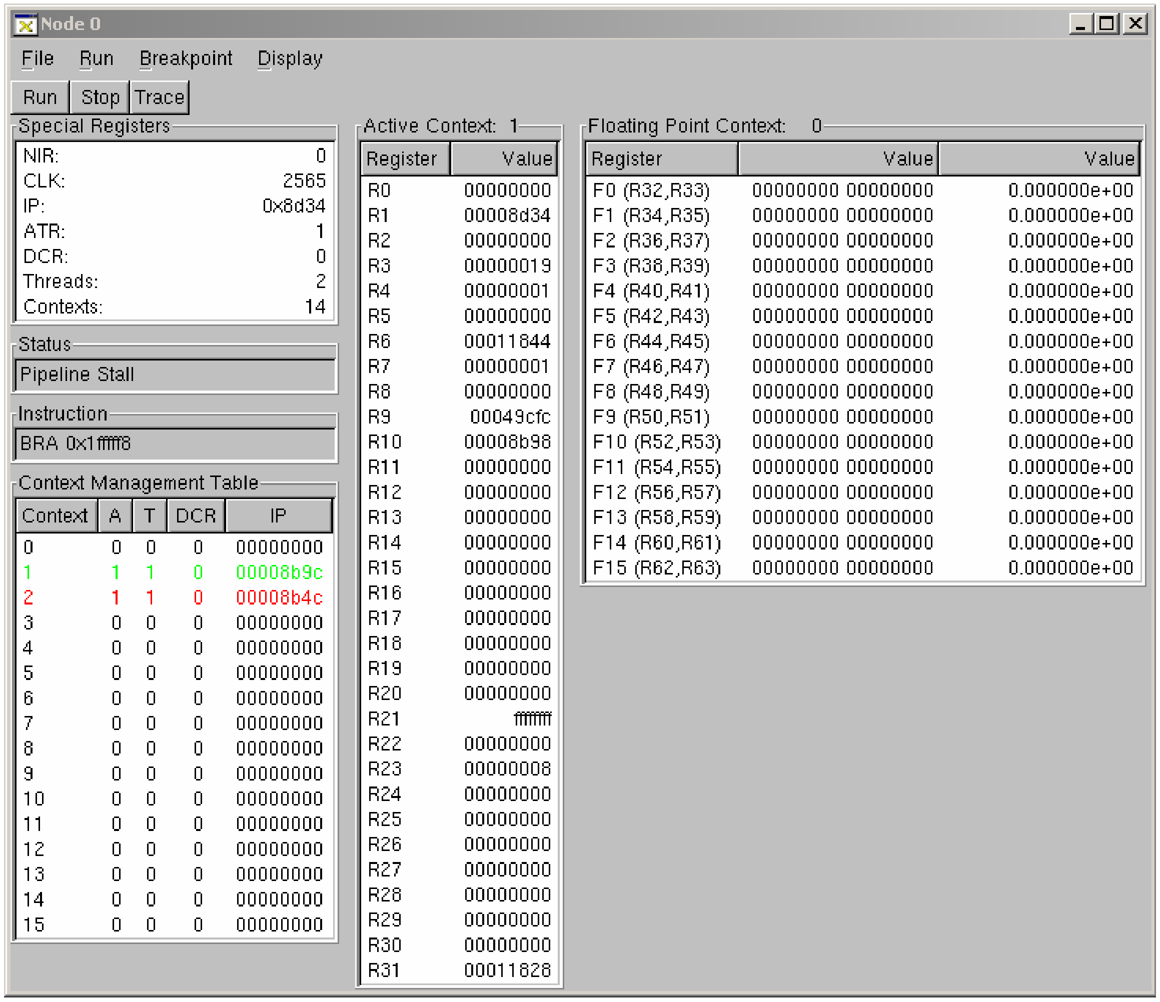


Figure 4.2: ***Screenshot of the graphical SCMP simulator with 64 processors.***

The simulator also includes an interface which displays detailed individual node information. Figure 4.3 shows the current state of a node along with the special registers including the Node ID (NIR), Clock (CLK), Instructions Pointer (IP), Active Thread

Register (ATR) and the Data Context Register (DCR). Execution of a program can be started or paused using this interface of the simulator.



Special Registers

NIR:	0
CLK:	2565
IP:	0x8d34
ATR:	1
DCR:	0
Threads:	2
Contexts:	14

Status

Pipeline Stall

Instruction

BRA 0x1ffffff8

Context Management Table

Context	A	T	DCR	IP
0	0	0	0	00000000
1	1	1	0	00008b9c
2	1	1	0	00008b4c
3	0	0	0	00000000
4	0	0	0	00000000
5	0	0	0	00000000
6	0	0	0	00000000
7	0	0	0	00000000
8	0	0	0	00000000
9	0	0	0	00000000
10	0	0	0	00000000
11	0	0	0	00000000
12	0	0	0	00000000
13	0	0	0	00000000
14	0	0	0	00000000
15	0	0	0	00000000

Active Context: 1

Register	Value
R0	00000000
R1	00008d34
R2	00000000
R3	00000019
R4	00000001
R5	00000000
R6	00011844
R7	00000001
R8	00000000
R9	00049cfc
R10	00008b98
R11	00000000
R12	00000000
R13	00000000
R14	00000000
R15	00000000
R16	00000000
R17	00000000
R18	00000000
R19	00000000
R20	00000000
R21	ffffffff
R22	00000000
R23	00000008
R24	00000000
R25	00000000
R26	00000000
R27	00000000
R28	00000000
R29	00000000
R30	00000000
R31	00011828

Floating Point Context: 0

Register	Value	Value
F0 (R32,R33)	00000000 00000000	0.000000e+00
F1 (R34,R35)	00000000 00000000	0.000000e+00
F2 (R36,R37)	00000000 00000000	0.000000e+00
F3 (R38,R39)	00000000 00000000	0.000000e+00
F4 (R40,R41)	00000000 00000000	0.000000e+00
F5 (R42,R43)	00000000 00000000	0.000000e+00
F6 (R44,R45)	00000000 00000000	0.000000e+00
F7 (R46,R47)	00000000 00000000	0.000000e+00
F8 (R48,R49)	00000000 00000000	0.000000e+00
F9 (R50,R51)	00000000 00000000	0.000000e+00
F10 (R52,R53)	00000000 00000000	0.000000e+00
F11 (R54,R55)	00000000 00000000	0.000000e+00
F12 (R56,R57)	00000000 00000000	0.000000e+00
F13 (R58,R59)	00000000 00000000	0.000000e+00
F14 (R60,R61)	00000000 00000000	0.000000e+00
F15 (R62,R63)	00000000 00000000	0.000000e+00

*Figure 4.3: **Screenshot showing the status of the processors, Register Table and the Context Management Table of the SCMP.***

To simulate a SCMP program using the SCMP simulator, the parameters and options are specified in a configuration file. This configuration file has the details such as the number of active processors, parameters to set the instruction cache value and the amount of memory to be included with every node. A sample configuration file is shown in Figure 4.4.

```
; Configuration file for median filter
; 64 nodes (8x8. All nodes active)
X:      8
Y:      8
; Instruction cache parameters
I_N:    1024
I_K:    1
I_L:    8

Num_Contexts: 16

; Metering specifications
meter_interval:         10000
meter_log_file:         median.meter.log
icache_file:            median.icache.log
instr_count_file:       median.instr.log
concurrency_file:       median.cpc.log
message_file:           median.msg.log
network_file:           median.network.log
;message_traffic_file:  median.msg_traffic.log
```

Figure 4.4: ***Sample SCMP simulator configuration file***

The options and values that can be specified in the configuration file is given in Table 4.1

Option	**Description**	**Default Value**
X: *value*	Number of nodes in the X dimension	8 (total number of nodes in the system is X*Y)
Y: *value*	Number of nodes in the Y dimension	8 (total number of nodes in the system is X*Y)
Mem_Size: *value*	Amount of memory included with each node, in bytes	1,048,576 bytes (1 MB)
Num_Contexts: *value*	Number of thread contexts (32 registers each) included with each node	16
I_N: *value*	Number of sets in the instruction cache	32
I_K: *value*	Number of lines per set in the instruction cache	1
I_L: *value*	Number of words per line in the instruction cache	8
meter_interval: *value*	Time interval (clock cycles) between updates to meter log files	1000
meter_log_file: *filename*	Summary log file information, printed at end of simulation	See section below for more information about log files
icache_file: *filename*	Log file for instruction cache, printed at end of simulation	
concurrency_file: *filename*	Log file for concurrency (number of active nodes), updated every meter interval	

message_file: *filename*	Log file for messages (number of messages and latency), updated every meter interval	
network_file: *filename*	Log file for network information (number of flits, throughput, and latency), updated every meter interval	
message_traffic_file: *filename*	Log file for collecting message traces, updated after every message reception	

Table 4.1: *Format of the configuration file*

The simulator also has an option that generates log files to measure the performance of the system. Some of the information is logged during the simulation of an application, and some of the files contain summary information that is written only at the end of the simulation. Each entry in the log file includes an average value, a minimum value, and a maximum value over that time interval. The different types of log files that the simulator can generate and its description is given below:

- ***meter_log_file*** : This file has the summary of the program execution. This includes such items as the number of threads, the average run-length of each thread, the number of messages, the average length, latency, and distance traveled for each message, etc.
- ***concurrency_file*** : This file contains details about the number of nodes that were active during each time interval. The file logs the average, maximum, and minimum number of active nodes during each clock cycle over the previous time interval.
- ***message_file*** : The message file has the information on number of messages in the network during each time interval. This includes the average, maximum, and minimum number of messages during each clock cycle over the previous interval,

as well as the average, maximum, and minimum message latency over the interval. The message latency is defined as the time from when the message header is injected into the network until the message tail is removed from the network.

- ***network_file*** : This file measures the SCMP network performance during each time interval. This includes the average, maximum, and minimum number of flits in the network; the average, maximum, and minimum latency of each flit; and the average, maximum, and minimum throughput of the network over the time interval. The latency is defined as the time from when the flit is injected into the network until it is removed from the network, and the throughput is defined as the number of flits removed from the network during each clock cycle.
- ***icache_file*** : This files has details about the performance of the instruction cache on each node. This file, written at the end of the simulation, has one line for each processor in the system. It lists the hit and miss rate for the instruction cache, as well as the number of capacity and compulsory misses.
- ***message_traffic_file*** : The traffic file has the information about each message sent during the simulation. Each entry in this file includes the message source and destination nodes, the time the message was sent, the time it arrived, the message type, and the message length.

4.4 Writing Parallel Programs

Before writing applications for the SCMP, various design aspects need to be taken into consideration. At the beginning of the execution of a program, Processor 0 is given the control first. Processor 0 has to create threads on other processors and also specify the name of the function that will be executed. Usually, after the initialization step of a program, a function is executed in parallel on all the processors by creating threads. This would involve creating threads on all the processors individually and then specifying the function name that is to be executed. Instead of the programmer having to create threads every time, the function *parExecute* of the SCMP library could be used. The *parExecute* function call by Processor 0, first creates threads on all the processors on the first row of the 2-D grid, and each of the processors in the row creates a thread on the processors in its respective column. Creating threads in this distributed manner reduces network congestion and leads to faster execution of the function call. *ParExecute* has the following syntax:

parExecute(num_nodes, function_name, parameters) where

- num_nodes – Number of nodes the function has to be executed on
- function_name – Name of the function that has to be executed
- parameters – Parameters passed to the function, separated with commas.

If a function has to be executed remotely by creating a thread on a different processor, the declaration has to be preceded by the *#PRAGMA remote* directive. The *#PRAGMA remote* directive before a function tells the compiler that the following function is executed remotely on a processor, and multiple instances of the same function will be executed on the processors. If a function is called within another function, then the *#PRAGMA remote* directive should not be included since the function is going to execute on the same thread and on the same processor.

Data communication is a one-way mechanism in the SCMP, i.e., there are send functions, which allow processors to send data to other processors, but there are no receive functions. Also, if a processor needs data from the local memory of another processor, then it has to create a thread message on the other processor and call a function to request the data. Since there is no signaling mechanism to inform a processor that data has arrived, the originating processor has to send synchronization data that will inform the processor that the data has arrived. Synchronization mechanisms in the SCMP are built with user-level software routines. Data messages sent from a processor A to a processor B are always received in-order. Thus, if processor A sends data and the synchronization data back to back to processor B, processor B knows that the data has arrived if the synchronization data has arrived. Synchronization is usually done by send a data value to a memory location, and the destination processor polling on that memory location at regular intervals. This is a one to one send and acknowledgement scheme used commonly in the SCMP. The data communication and synchronization between two nodes is shown in Figure 4.5.

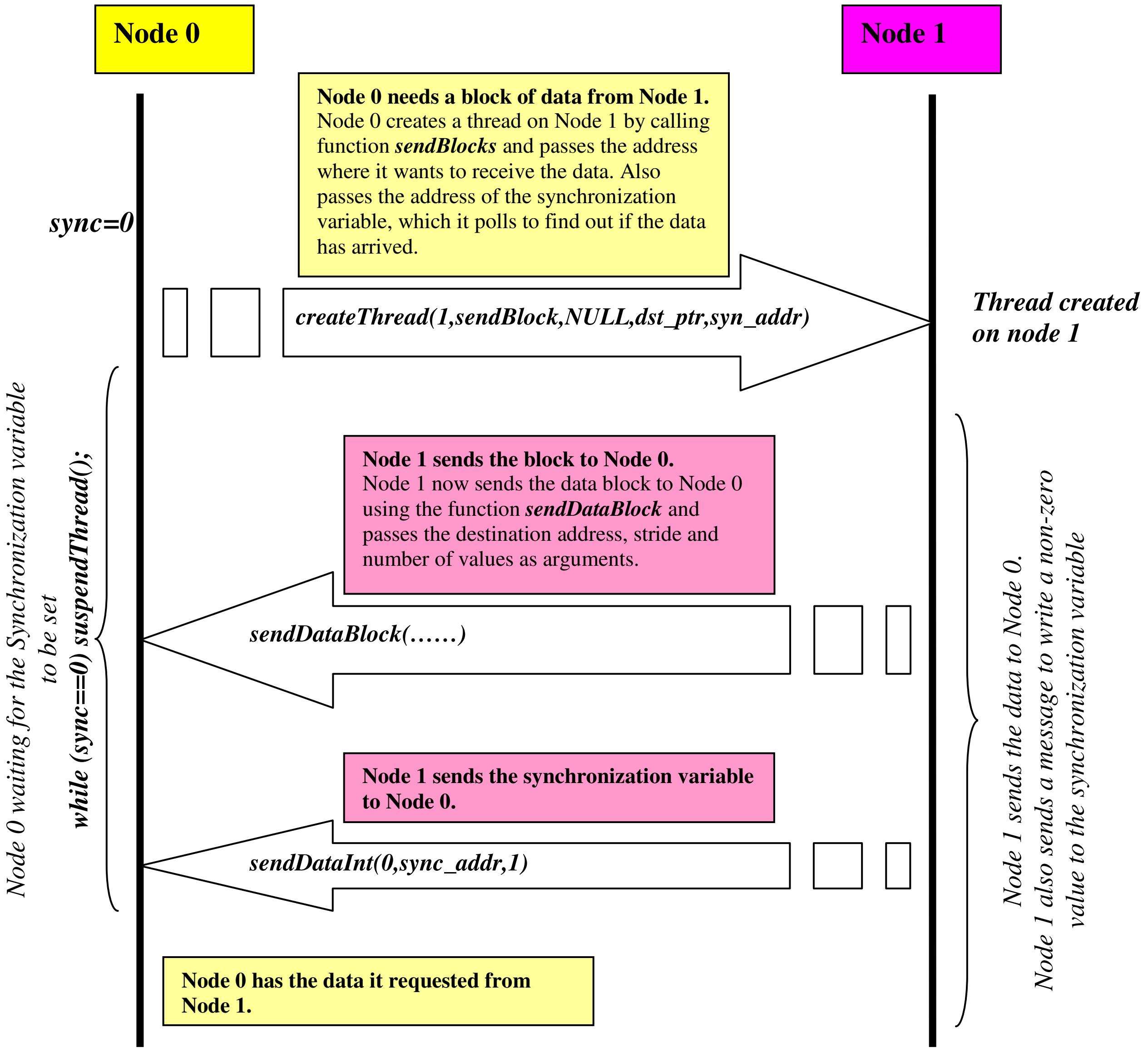

Node 0
Node 1
Node 0 needs a block of data from Node 1.
Node 0 creates a thread on Node 1 by calling function sendBlocks and passes the address where it wants to receive the data. Also passes the address of the synchronization variable, which it polls to find out if the data has arrived.
sync=0
createThread(1,sendBlock,NULL,dst_ptr,syn_addr)
Thread created on node 1
Node 0 waiting for the Synchronization variable to be set
while (sync==0) suspendThread();
Node 1 sends the block to Node 0.
Node 1 now sends the data block to Node 0 using the function sendDataBlock and passes the destination address, stride and number of values as arguments.
sendDataBlock(……)
Node 1 sends the synchronization variable to Node 0.
sendDataInt(0,sync_addr,1)
Node 1 sends the data to Node 0.
Node 1 also sends a message to write a non-zero value to the synchronization variable
Node 0 has the data it requested from Node 1.

Figure 4.5 : *Data communication process in the SCMP*

Another common synchronization operation that is used in various applications is a barrier. A barrier forces all processes to wait until all the processors have finished the execution of a particular task. Upon completion of this task, the processors proceed to the next instruction in the program. In the SCMP, a 2-dimensional tree-based barrier is used to distribute the communication load. A variable is passed to declare the barrier in a program and every processor polls on this variable to check if the barrier task has been accomplished or not. Once all the processors have incremented the barrier variable, the processor knows that the entire barrier task is completed. While a processor is polling on a variable, it can suspend its current thread and continue the execution of the thread when its turn arrives.

SCMP allows global variables to be declared for all the processors. Global variables are not shared memory variables, as the name implies. Instead they are variables declared for all the processors, each of them having the same reference name. Global variables are mapped to the same address on each node. Dynamically allocated memory, however, may be at different addresses on different nodes.

5. Application Benchmarks

5.1 Smith – Waterman Sequence Alignment

Bioinformatics is the use of computers in solving information problems in life sciences, mainly, involving the creation of extensive electronic databases on genomes, protein sequences etc. Secondarily, it involves techniques such as three-dimensional modeling of biomolecules and biological systems. The transformation and manipulation of data requires a lot of computational power and resources. It is becoming more and more common to use faster and state-of-the-art computers for these computations. A very common problem in the field of bioinformatics is the comparison of a gene sequence to a database of sequences. The rapidly increasing amounts of genetic sequence information available present a constant challenge to hardware and software developers. When looking for sequences in a database similar to a given query sequence, the search program computes an alignment score for every sequence in the database. This score represents the degree of similarity of the query sequence with the sequences in the database. The algorithm for computing the score of a matrix is given in the subsequent section. Current database searching algorithms such as FASTA [www.ebi.ac.uk/fasta33] and BLAST [www.ebi.ac.uk/blast2], implement the Smith-Waterman sequence algorithms but the sacrifice result validity for speed. As a result, these algorithms may not accurately predict distantly related sequences. The parallel implementation of the Smith-Waterman algorithm calculates the sequence distances accurately using the SCMP system.

5.1.1 Smith – Waterman – Algorithm

Consider, two sequences, V = $v_1 \ldots v_n$ and W = $w_1 \ldots w_m$. Figure 5.1 represents the similarity matrix for sequences V = ATCTGAT and W=TGCATA. A dynamic programming algorithm can be used to compute the similarity, S(V, W).

For all $1 \leq i \leq n$ and $1 \leq j \leq m$, S(i,0) = S(0,j) = 0. Then , S(i,j) is computed with the following recurrency:

$$S(i,j) = \max \begin{cases} \mathbf{S(i-1,j)} \\ \mathbf{S(i, j-1)} \\ \mathbf{S(i-1,j-1) + 1 , \text{ if } Vi = Wj} \end{cases}$$

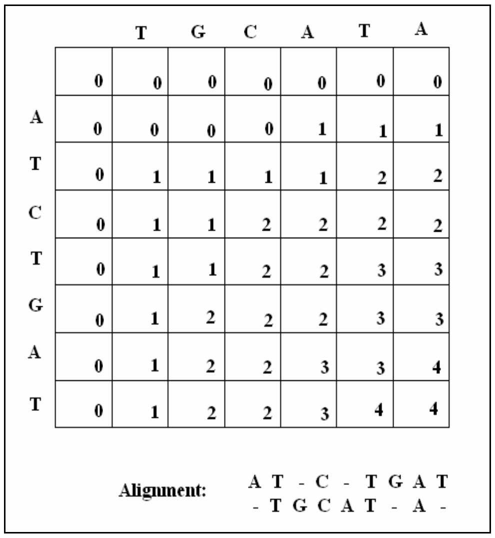
The edit distance, d(V,W) can be calculated from the similarity, S(V,W).

$$\mathbf{d(V, W) = n + m - 2\,S(V, W)}$$

The algorithm specified above is for global sequence alignment. The algorithm for local sequence alignment includes a small change in the recurrency:

$$S(i,j) = \max \begin{cases} \mathbf{0} \\ \mathbf{S(i-1,j)} \\ \mathbf{S(i, j-1)} \\ \mathbf{S(i-1,j-1) + 1 , \text{ if } Vi = Wj} \end{cases}$$

The calculation of the S(i,j) elements of the output matrix represents the Smith-Waterman algorithm.



		T	G	C	A	T	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
G	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
T	0	1	2	2	3	4	4

```
Alignment:   A T - C - T G A T
             - T G C A T - A -
```

Figure 5.1: *Example of a Similarity Matrix, S(V, W)*

The goal of the algorithm is to come up with the similarity matrix as shown in Figure 5.1. This similarity matrix is the backbone of various other complicated applications in bioinformatics.

5.1.2 Smith – Waterman Design Considerations

The algorithm has to generate all the values in the matrix. Hence, the complexity of the algorithm is O(n x m). Consider, the data dependencies for matrix element S(i,j). The dependency graph is shown in the Figure 5.2. The data dependencies exist only on top and to the left of each cell. Thus the diagonal values of the matrix can be computed concurrently. The parallelized algorithm computes up to 'n' operations at each step.

$$\begin{tikzcd}
S(i-1,j-1) \arrow[dr] & S(i-1,j) \arrow[d] \\
S(i,j-1) \arrow[r] & S(i,j)
\end{tikzcd}$$

Figure 5.2: *Data dependency graph*

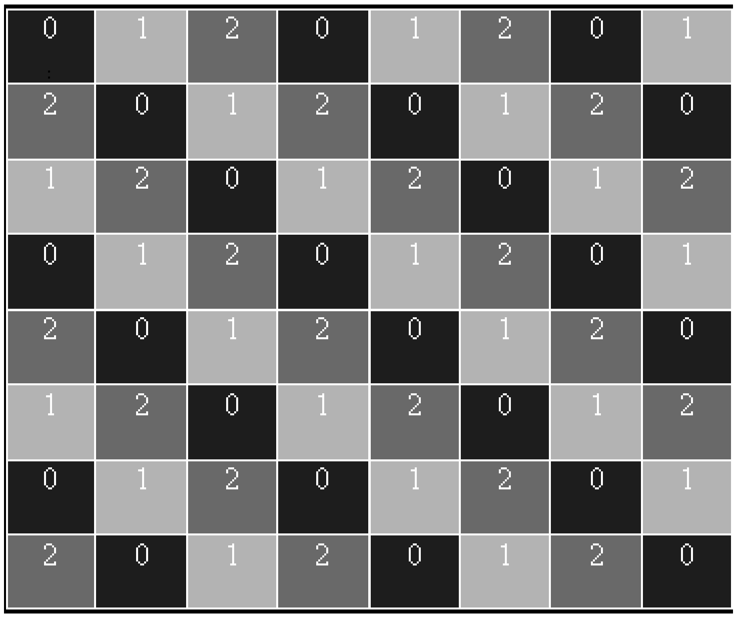
If ‘n’ processing elements are available, the matrix can be computed in ‘n + m’ steps.

This is the most generalized way to parallelize the Smith-Waterman Algorithm.

When considering the application of SCMP to this problem, the algorithm was written in such a way that it optimally utilized the parallel and message-passing feature of the SCMP.

5.1.3 Smith – Waterman – Implementation

The output matrix to be computed can be divided into blocks of equal sizes mapped to different processors. As the blocks are processed, the data that is required for other blocks is passed via messages sent over the network. The data dependency graph in Figure 5.2 shows that element S(i-1,j-1) has to be computed before S(i,j) can be calculated. In terms of blocks, this is illustrated in Figure 5.3. The computation initially starts at the top-left corner block and as the matrix is computed, the algorithm progresses along the diagonal blocks. Every internal block of the output matrix requires the result from three other blocks. Thus, this algorithm initially starts slow, but then as more blocks are computed and data is propagated, it achieves a very high speed-up. Towards the end of the computation, fewer processors are active and fewer calculations are required.



0	1	2	0	1	2	0	1
2	0	1	2	0	1	2	0
1	2	0	1	2	0	1	2
0	1	2	0	1	2	0	1
2	0	1	2	0	1	2	0
1	2	0	1	2	0	1	2
0	1	2	0	1	2	0	1
2	0	1	2	0	1	2	0

Figure 5.3: *Cyclic assignment of blocks to processors*

As soon as a block completes its calculations, it has to send its data to the block on its right, and also to the block below it. The other block can start its computation as this data is received. Figure 5.4 shows how the algorithm progresses as blocks finish their computations [Yang02].

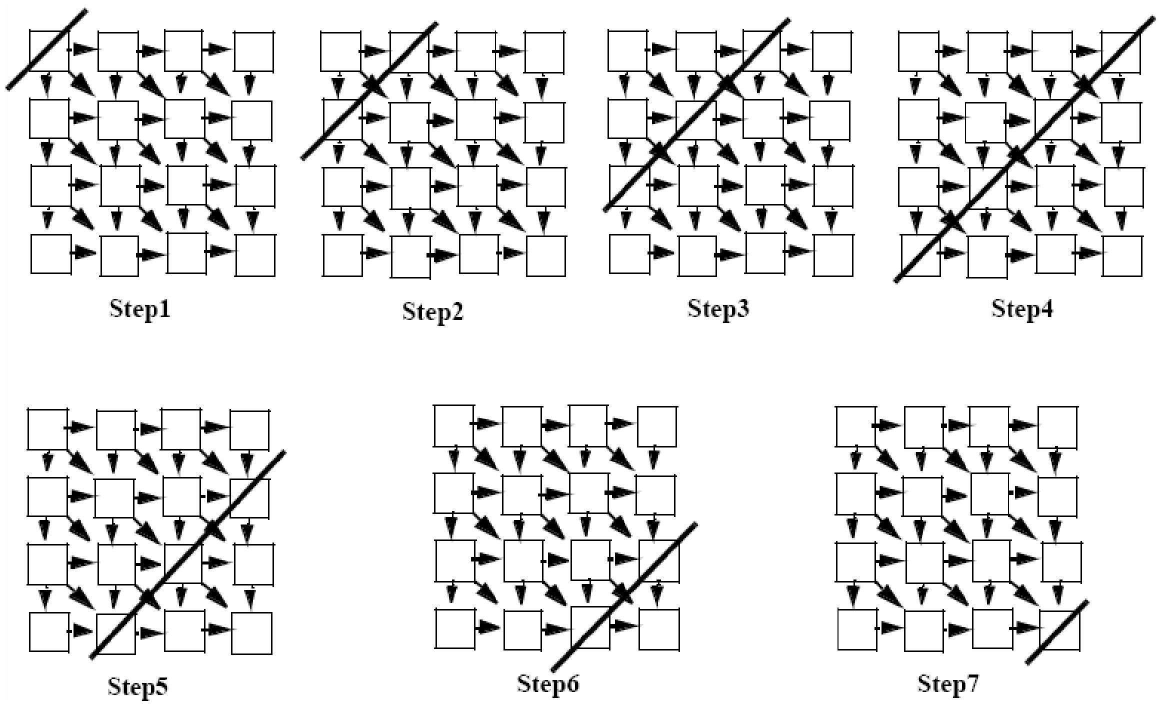


Figure 5.4: *Wavefront propagation of computation amongst blocks*

5.2 Radix Sort

The integer radix sort [Peb03] is a kernel benchmark that is widely used in various applications such as database management system, operating system and various aerospace and ocean applications. Radix sort requires bulk data transfer (keys) from one processor to another. Radix sort is a multiple pass sort algorithm that does not do comparison of keys like the common sorting algorithms. Instead, it relies on the representation of keys as b-bit integers. The algorithm resembles the way humans normally analyze information and therefore is simple to understand. A radix sort example [Radix00] with 8 keys with each key having 3 digits is illustrated in Figure 5.5.

5.2.1 Radix Sort - Algorithm

The basic radix sort algorithm examines the keys to be sorted r bits at a time. The keys are sorted beginning with the least significant digit, and the sort continues till the most significant digit is sorted. During this sorting, each key is placed in a bucket and a first-in first-out list is created. After each pass, the keys are collected and the next least significant part is sorted. This is continued until all the parts are sorted. Collecting the keys after the last pass will result in a sorted list of the input keys.

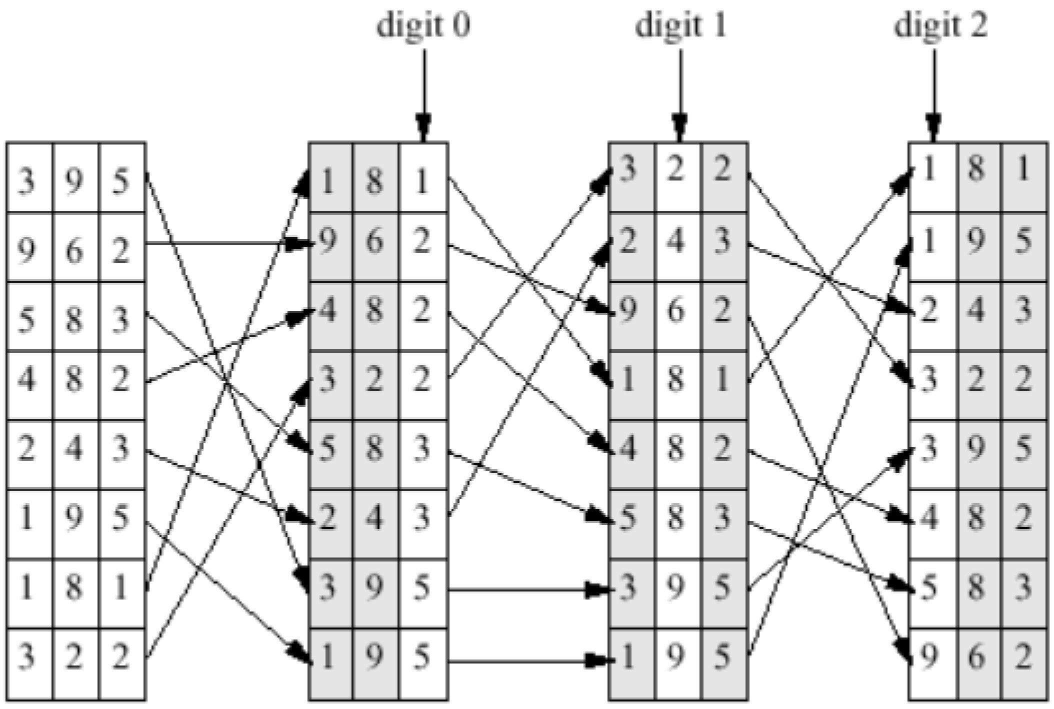


Figure 5.5: *Radix sort example with 8 keys*

It is evident from the radix sort algorithm that it has a complexity of O(*kn)* where k is the length of the number of digits and n is the radix. Radix sort gives the best performance when the number of keys increases with respect to the length of each key.

5.2.2 Radix Sort – Design Considerations

The radix sort algorithm implemented is based on the radix sort implementation of the SPLASH-2 benchmarks. The algorithm had to be converted for a message-passing system since SPLASH-2 was implemented for shared memory multiprocessor system. The SCMP is a matrix of 64 processors, with each processor having its own memory. Thus, the algorithm needs to be converted from a shared memory to a local memory message-passing algorithm. This meant that each processor would be allocated a memory space to hold only a set of keys assigned for that processor. The parallel radix sort algorithm that was implemented is given below:

a. Make a pass over the local n/p keys to build a local histogram of key value densities. If a key encountered has the value i in the current phase, then the ith bin of the histogram is incremented.

b. When all the processors have completed Step a, then the processors send the local histogram to processor 0, which acts as a master processor. The global histogram keeps track of both how many keys there are of each value for the digit, and also each of the processors from where the keys arrived. Processor 0 then calculates the global histogram from the information received, and then sends the information to all the processors.

c. Now locally update the ranks of the keys of the current processor by reading the global density array received from processor. Do a permutation / combination according to the global ranks and then send the keys to those processors accordingly.

d. Go to step a and repeat for the next least significant digit

The algorithm also requires space to be allocated for a histogram and a global density array. The histogram is an array of the size of the radix and each entry of the histogram has a count of the number of keys present locally with each radix. Another array of size of the radix is created to store new locations of the keys once Processor 0 receives the all the histograms. The local keys are then transferred between processors, and every processor receives new keys. These keys are in sorted order of the radix under computation.

5.2.3 Radix Sort – Implementation

The basic idea for the parallel implementation of radix sort is to first divide the keys evenly among all the processors. These keys are stored in a global array that is declared on all the processors. This global array exists on all the processors, which results in every processor having a local copy of the part of the input array. The partition occurs in such a way that processor 0 is assigned the first set of n/p keys, processor 1 the next n/p set of keys, and so on. Each of the processors then generates its local histogram. The value of the histogram at index *i* contains the number of digits having the value *i*. During each iteration of the radix sort algorithm, the local ranks are sent to Processor 0 and then it computes the global ranks. These global ranks are sent to all the processors. Once all the processors receive the global rank, the keys are sent to the new locations. Thus, this algorithm requires two barriers, one before receiving the global rank, and one after sending the keys to the processors. An example of the parallel radix sort algorithm is given in Figure 5.6, with each communication step explained. The example sorts a set of 20 keys using 5 processors.

Number of Processors – 5
Number of Keys – 20
Radix – 4

Processor 0 (P0)	Processor 1 (P1)	Processor 2 (P2)	Processor 3 (P3)	Processor 4 P(4)
3	1	3	1	1
0	0	0	0	2
2	3	1	3	1
2	1	0	3	0

This example illustrates the parallel radix sort algorithm with radix 4. There are 5 processors and the 20 keys are to be sorted. The initial assignment of the keys is given on the left, with each processor having 4 keys.

Local count of each bin

Bin #	P0	P1	P2	P3	P4
0	1	1	2	1	1
1	0	2	1	1	2
2	2	0	0	0	1
3	1	1	1	2	0

Each processor then generates a local histogram of the number of keys belonging to each bin. Here, Processor 0 had 1 key with value 0, 2 keys with value 2 and 1 key with value 3. Similarly, other processors also generate their histograms.

Global Density Matrix after sending keys to P0

Radix Value	P0	P1	P2	P3	P4
0	0	1	2	4	5
1	6	6	8	9	10
2	12	14	14	14	14
3	15	16	17	18	20

A global density matrix, which lets every processor know the rank of the keys it has compared to the other processors is computed.

Local Update of Ranks
Key Value - Global Rank

P0	P1	P2	P3	P4
3 – 15	1 – 6	3 – 17	1 – 9	1 – 10
0 – 0	0 – 1	0 – 2	0 – 4	2 – 14
2 – 12	3 – 16	1 – 8	3 – 18	1 – 11
2 – 13	1 – 7	0 – 3	3 – 19	0 – 5

The new positions are assigned to the keys and the key values are sent to the corresponding new processors.

New locations of the keys after communication
Key Value – Global Rank

P0	P1	P2	P3	P4
0 – 0	0 – 4	1 – 8	2 – 12	3 – 16
0 – 1	0 – 5	1 – 9	2 – 13	3 – 17
0 – 2	1 – 6	1 – 10	2 – 14	3 – 18
0 – 3	1 – 7	1 – 11	3 – 15	3 – 19

The sorted list after the key values have been sent to the processors.

Figure 5.6: *Parallel radix sort example*

5.3 Image processing

Two image-processing algorithms were implemented to run on the SCMP processor, Median filter and Sobel's edge detection. Median filtering is a method of smoothing images, i.e. reducing the amount of intensity variation between neighboring pixels in a grayscale image. It is often used to reduce the noise in an image and usually does a better job than a mean filter. The median filter algorithm has its application in image processing within industrial automation and also automated visual inspection. One such industrial automation example is for analyzing predetermined features of parts and look for defects in the manufactured parts. Sobel's algorithm is a gradient-based edge detection algorithm used to find the edges of an image. In this edge detection method, the normal assumption is that edges are the pixels with a high gradient. A fast rate of change of intensity in the picture results in better output edge image quality. The sobel edge detector uses a pair of 3x3 convolution masks, one determining the gradient for the x-direction and the in the y-direction. The following are the actual Sobel convolution masks:

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Figure 5.7: *Sobel's convolution masks*

Both the median filter and the sobel's edge detection algorithm were implemented in a similar manner. Both the algorithms use a 3x3 window of values around a pixel for their computations. Thus, both the algorithms had similar data layouts and communication patterns. The only difference between the two was the way the image was processed. The convolution masks were used to calculate the new image for Sobel's edge detection.

5.3.1 Median Filter – Algorithm

The median filter algorithm examines each pixel in an image and then looks at its eight neighboring pixels to decide where it fits in the surroundings or not (Figure 5.8). It then replaces its values by the median of its neighboring pixel values. Since the image is made up of pixel, and each pixel has a corresponding ASCII value, it is possible to compute the median with just a few simple calculations.

121	122	123	125	121
111	112	113	115	114
117	112	**145**	114	117
110	110	110	111	121
132	123	111	123	123

Value – 145

Neighborhood Values – 112, 113,115, 112, 114, 110, 110 and 111

Median Value – 112

Figure 5.8: *Calculating the median of a pixel neighborhood. As it can be seen, the pixel being operated on has an unusually high value compared to its neighbor. The pixel value of 145 is replaced with the median value 112.*

5.3.2 Median Filter – Design Considerations

One way to parallelize this algorithm would be to divide the image equally in blocks to the every processor and then do the communication by sending each of its corner column and row to its neighboring processor. This would require every block to receive two extra rows and two extra columns since the algorithm requires the pixels eight neighboring values. But, memory in the SCMP is word aligned. Thus, to send just one pixel value, the whole word value should be sent and received by the other processors. Also since the image is stored in a two-dimensional array in row major format, the column value would have to be sent individually, and thereby sending a lot of unnecessary image data. Instead of assigning image blocks to processors, the image was divided into rows, and a processor was given an array of rows to compute. This assignment of the image would every processor to just send its top most and bottom most rows to the processors above and below it, respectively. The row blocking would require just two send operation and two receive operations for every processor. Also, a block of memory could just be read directly and sent to a different processor with the row wise assignment of the image.

5.3.3 Median Filter – Implementation

The median filter was implemented by assigning a block of rows to every processor and then every processor applies the algorithm to the image piece it owns (Figure 5.9). At the beginning of the program, every processor reads its part of the image. There is a barrier after this since row transfers take place after this operation. Every processor needs a the top row from the processor above it, which has the bottom part of the image, and also requests the bottom row from the processor, which has the top part of the image. To accomplish this task of transferring rows between processors, each processor creates a thread on the processors that own the top and the bottom rows, and the thread executes a function which sends the row to the requesting processor. After this transfer takes place, all the data required apply the median filter algorithm is present in the every processors individual memory. This is the only communication step required for the algorithm. There is no extra overhead associated with this other than the extra memory required to hold the top and the bottom row for every processor.

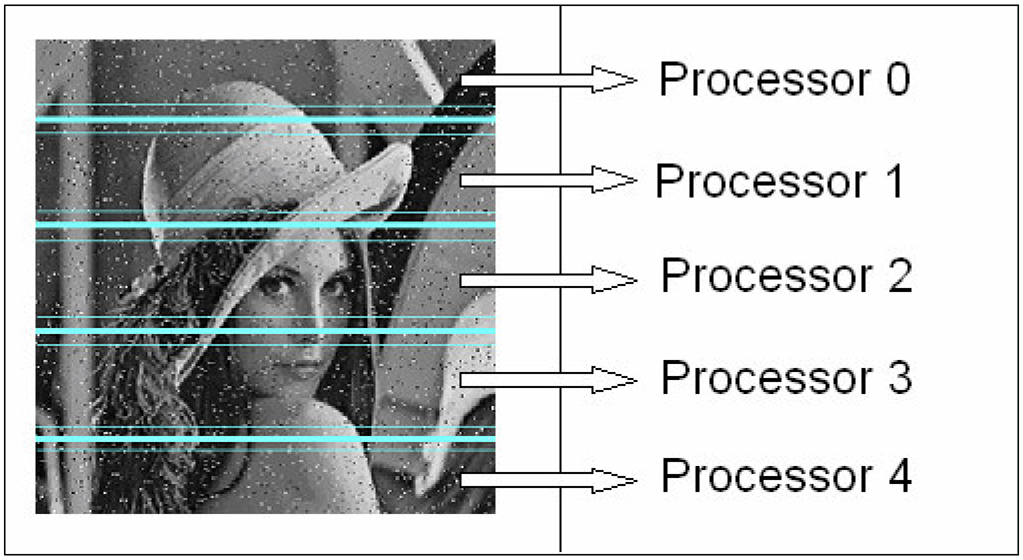


Figure 5.9: *Sample distribution of image amongst processors*

Once the algorithm is applied and the new image is obtained, each processor writes the updated data to an output file. This output file is written at various offset location such that the image data between any two processors do not overlap. The program finishes execution after every processor has finished writing its data to the output file. Figures 5.10 and 5.11 show sample results from the two image processing algorithms implemented.

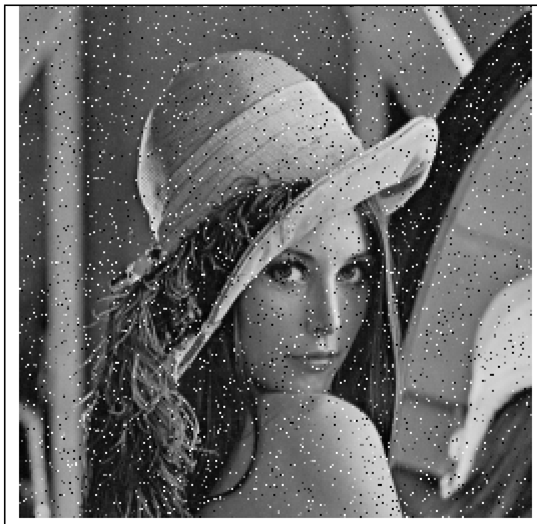
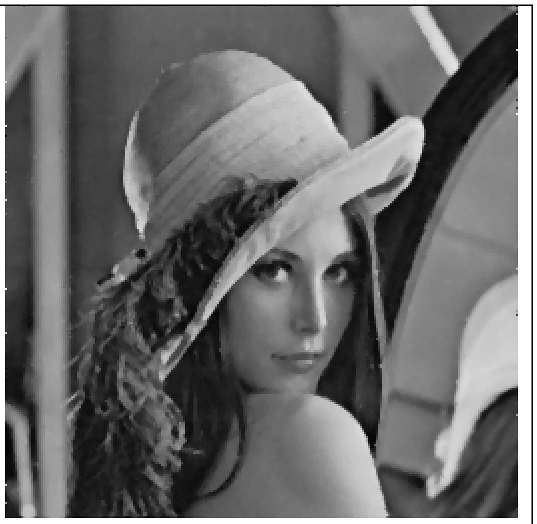
	

Image with salt and pepper noise	*Image after applying the median filter*

Figure 5.10: *Image with noise and image after applying the median filter*

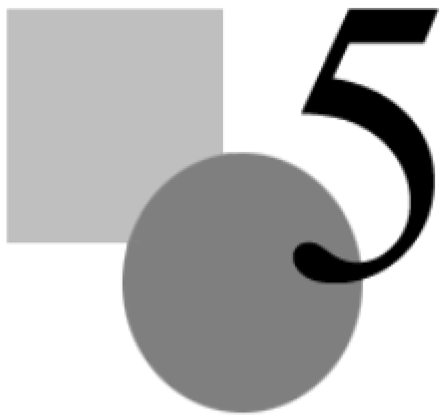
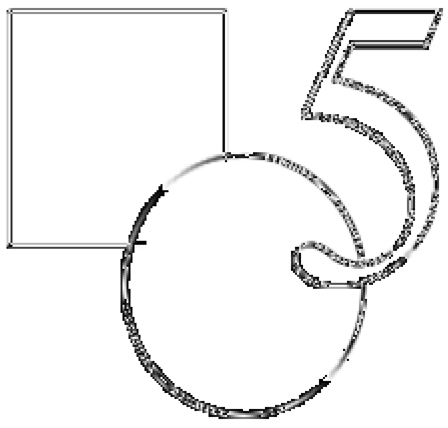
	
Original Image	*Image after Sobel's edge detection filter*

Figure 5.11: *Original image and image after applying the Sobel's filter*

5.4 LU Factorization

LU factorization is a Gauss-Seidel elimination method for solving a system of linear equations, eliminating one variable at a time by subtracting rows of the matrix from scalar multiples of other rows of the matrix. It involves decomposing an n x n matrix A into a product of the lower triangular matrix (L) and the upper triangular matrix (U). Thus

$$LU = A$$

A lower triangular matrix has all the elements above its diagonal as zero (Figure 5.12a). An upper triangular matrix has non-zero elements in the upper part of its diagonal (Figure 5.12b). After decomposing a matrix into its L and U components, the resultant matrix can be used to solve a set of equations.

$L = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}.$	$U = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}.$
Figure 5.12a *Lower Triangular Matrix*	***Figure 5.12b*** *Upper Triangular Matrix*

Figure 5.12: *Lower and Upper Triangular Matrices*

The utility of this algorithm is to solve linear systems of equations, and it is also used in various scientific applications as well as optimization methods such as linear programming. It is a well-structured computational kernel that is not trivial yet familiar and easy to understand and implement [Golub97].

5.4.1 LU Factorization – Algorithm

The algorithm described here is useful for factoring an n x n matrix. The matrix is divided into B-by-B blocks. By dividing the matrix into B-by-B blocks, data can be reused and temporal locality can be exploited. After the matrix is divided into blocks, the algorithm treats the matrix as consisting of n/B-by-n/B blocks rather than n-by-n elements. Blocking helps to reduce the communication overhead between processors. If blocking was not implemented, then during the calculation of every pivot point, individual messages would have to be sent to its corresponding row and the column, and also to all inner matrix element. By blocking, these messages are aggregated into an entire block of data and the data can be transferred using a single message. This leads to a faster execution of the program due to the increase of the computation to communication ratio a. An example of how the LU factorization is done in a 3x3 matrix is shown in Figure 5.13 [Moore99].

Find the L and the U matrix of the given matrix

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

Step 1. Reduce the matrix using the Gaussian Elimination method, such that all the elements below the diagonal are zeros. This is the upper triangular matrix (U).

$$\mathbf{U} = \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 0 & 0 & 18 \end{bmatrix}$$

Step 2. To obtain the lower triangular part of LU factorization, form a matrix from the multipliers,i.e what we multiplied each row by when we performed the reductions, with ones along the diagonal.

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 5 & 1 \end{bmatrix}.$$

Step 3. If we multiply the two L and the U matrices, it returns the matrix A.

$$\mathbf{LU} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 5 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -5 \\ 0 & 0 & 18 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{bmatrix} = \mathbf{A}$$

Figure 5.13: *LU factorization example*

5.4.2 LU Factorization – Design Considerations

As discussed in the algorithm, it is evident that the algorithm will perform the best when the input matrix is divided into blocks. Without blocking, a processor would have to compute an element, then compute the element on the right and continue doing so until it reaches the next pivot element. When it reaches the pivot element of the next row, it has read the data equivalent to an entire row of the matrix, and this could hurt in very poor performance measures. Thus the blocked version was implemented in parallel for the SCMP. Every processor is assigned one or more blocks in a cyclic order of assignment of blocks. The maximum size of a block should not be more than the memory every processor has. This is due to the fact that it would be faster for a processor to get a whole block from its own memory and not requesting a part of the block from another processor.

5.4.3 LU Factorization – Implementation

A blocked version of the LU factorization was implemented for the SCMP. The input matrix is divided into user defined blocks of size B-by-B. Thus, the input n-by-n matrix is divided into (n/B)-by-(n/B) matrix of blocks. The user also specifies the number of processors that are required for the LU factorization.

Blocked Dense LU Factorization

```
1. For $k$=0 to $N-1$ do
    2. Factor diagonal block $A_{kk}$
    >>> Barrier <<<
    3. Update all perimeter blocks in Column $k$ and Row $k$ using $A_{kk}$
    >>> Barrier <<<
    4. For $j=k+1$ to $N-1$ do
        5. For $i=k+1$ to $N-1$ do
            6. $A_{ij} = A_{ij} - A_{ik} * A_{kj}$
```

A block (I,J), is assigned to a processor using the formula:

*(J % num_cols) + (I%num_rows)*num_cols*

num_cols – Number of processors in the Y dimension.

num_row – Number of processors in the X dimension.

After assigning blocks to processors, the main LU factorization function is called. The function loops over all the diagonal blocks of the matrix, and first factorizes the diagonal block. A barrier is implemented after this operation since all the processors have to wait for the diagonal block to be factorized first. After factorizing the diagonal block, the block is sent to all the blocks in the bottom of its column, and all the blocks to the right of its row. Once the blocks receive the factorized pivot block, they divide it by the pivot block. Again, there is a barrier after this since all the other blocks have to wait for the division step. Once the barrier is passed, all the blocks of the inner matrix need to be updated. Then, the program looping over all the blocks updates the inner matrix. Every block of the inner matrix requests the blocks it requires the next update step. This is done by creating threads on processors that own the required block, and then requesting the processor to send block to its memory buffer. After receiving the blocks, each of the inner matrix blocks is updated. There is a barrier after this too, since all the processors need to be synchronized before the start of the next iteration and factoring the next pivot block. Figure 5.18 illustrates the LU factorization progress as blocks are computed.

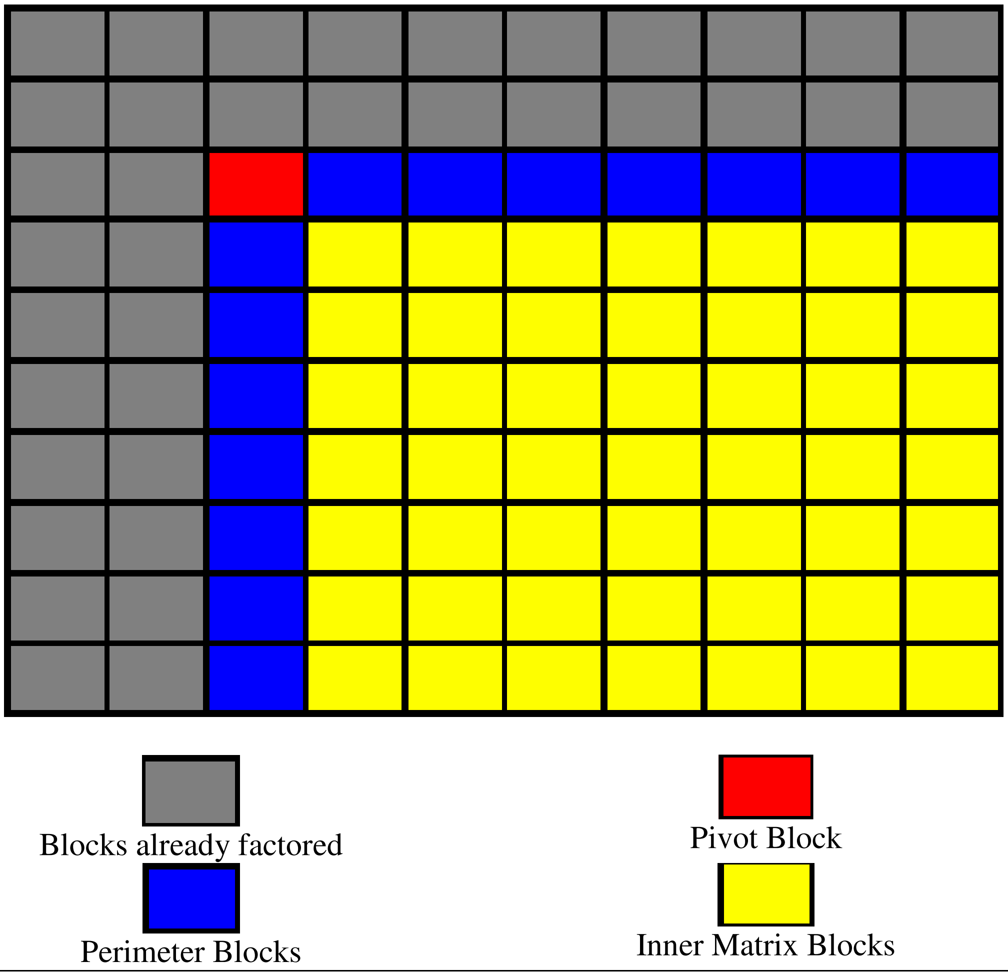


Figure 5.14: *Blocked LU Factorization algorithm progress matrix*

6. Analysis and Results

All the benchmarks and applications were run on the SCMP system simulator. The simulator output gives information such as the number of cycles and the number of instructions, which can be used to compare the results of the application execution with other existing systems.

The speedup of a program is the time it takes the program to execute on a single processor versus the execution time on multiple processors. For the SCMP, we consider the number of cycles it takes for a single processor to run the program. Speedup is the time it takes to execute a program in serial divided by the time it takes to execute the program in parallel.

$$\textbf{Speedup S} = \mathbf{T}_{\textbf{Single Processor}} / \mathbf{T}_{\textbf{Multiple Processors}}$$

To evaluate the parallel performance of a given benchmark, the benchmark was first run sequentially and the simulator output was studied. Then the parallel version was executed multiple times, varying the number of active processors. The two results were plotted and the results were studied.

The performance of SCMP was also compared with a 2.5 GHz Intel P4 processor. For this comparison, it was assumed that the SCMP processors are also running at 2.5GHz. The results of this comparison are also discussed and showed along with individual benchmark application results.

6.1 Smith-Waterman Sequence Alignment

As discussed earlier, the blocked version of Smith-Waterman sequence alignment algorithm was implemented for the SCMP. As more and more blocks are computed, the algorithm progresses at a faster rate. Towards the end, fewer blocks are computed and then the algorithm comes to a slow end. The speedup observed for the blocked version was very close to the ideal speedup. This is because the amount of data that is transferred through the network is small, with just a few values for every block calculation. Also, since every processor had the part of the input sequence it had to compute, there was not communication required for the input sequence. The speedup for the algorithm with a sequence size of 256 is given in Figure 6.1.

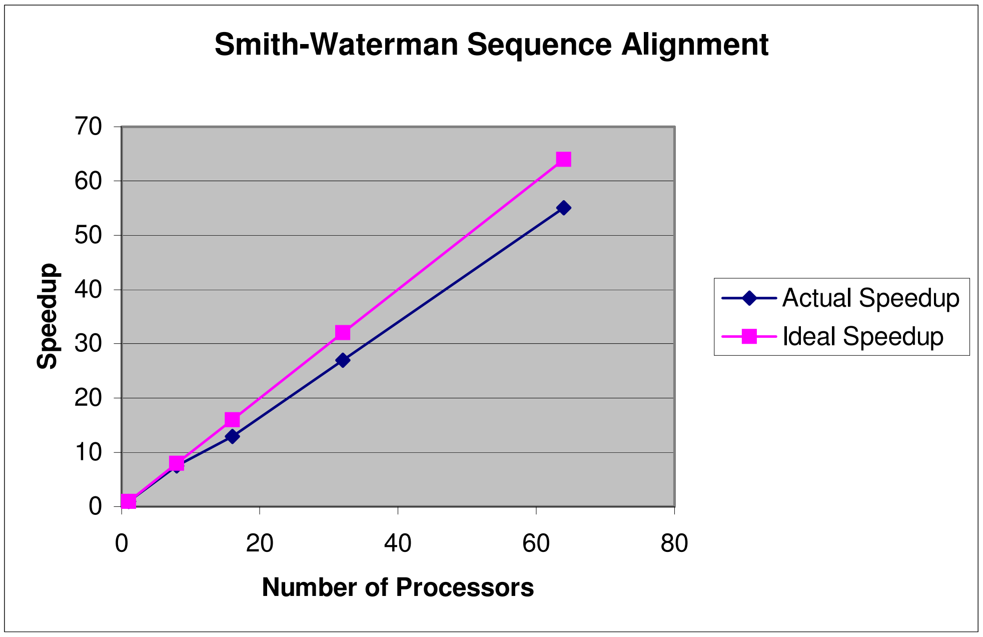


Figure 6.1: *Smith-Waterman speedup for a sequence of size 256*

The performance of the Smith-Waterman algorithm on a 2.5GHz Pentium 4 machine is compared to the SCMP processor. This comparison is shown in Figure 6.2.

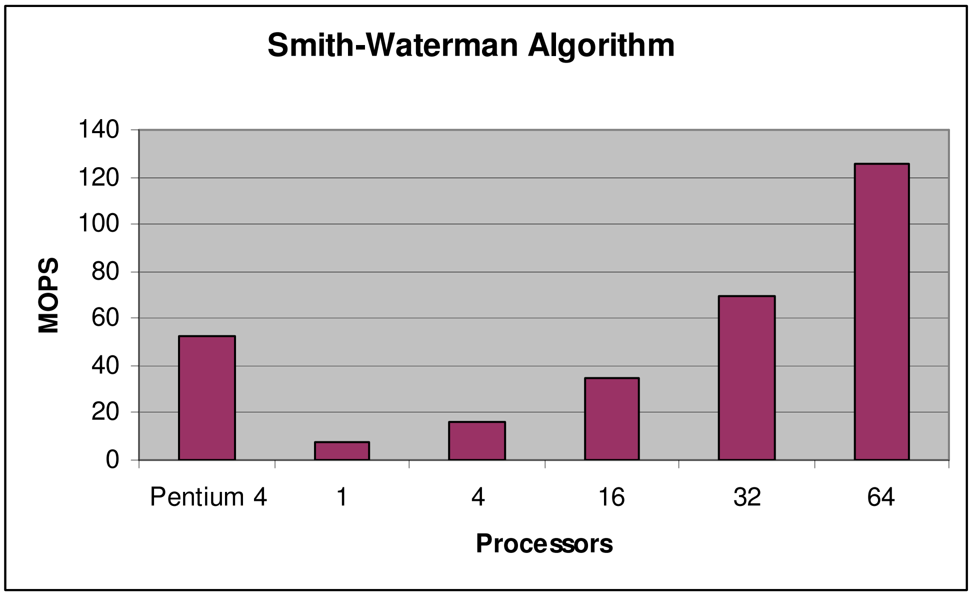


Figure 6.2: *Performance of SCMP vs. Intel Pentium 4 for the Smith-Waterman Algorithm*

From the graph in Figure 6.2 we can see that MOPS rating for the Pentium 4 machine is much higher than the SCMP. This is due to a high communication to computation ratio for the algorithm. But as larger blocks are computed, this ratio will get smaller, since only the last row and the last column needs to be sent to other processors.

To compute the sequence alignment for a matrix, each element in the sequence would be compared with every element in the other sequence. Each of these comparisons would require three calculations to find the maximum alignment score. Thus the total number of operations required for a sequence of size n for calculating the similarity matrix is **$3n^2$.** Thus, for a 256-by-256 input sequence, the number of operations required would be 196608.

6.2 Radix sort

The performance of radix sort is strongly dependent on the choice of the radix. A larger radix would result in fewer numbers of digits in a number where as a smaller radix would require more digits to represent the same number. The fewer the number of digits, the fewer are the iterations in the radix sort algorithm. A larger radix is not always efficient since it would also require a larger histogram and a larger density array to be allocated for every processor. The speedup graph for a data set of 4096 keys with radix 10 is given in Figure 6.3

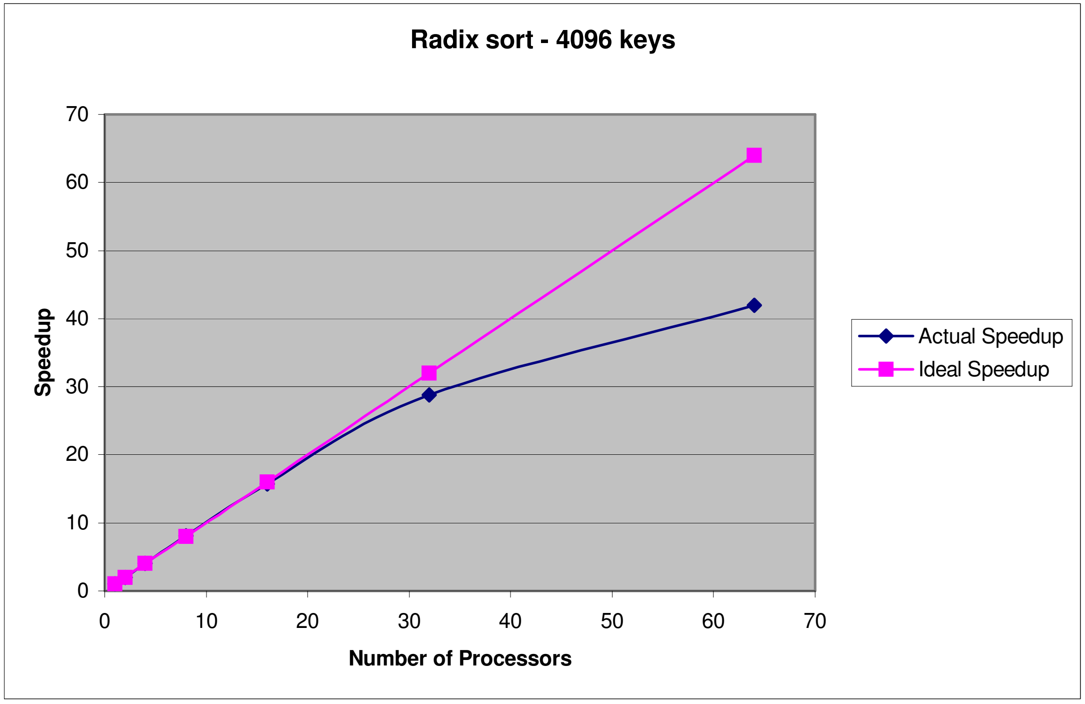


Figure 6.3: *Radix sort speedup*

As it can be seen from the graph, the speed up is not linear as compared with the sequential execution of the program. This is due to the overhead involved during the transfer of keys at the end of each iteration. The computation to communication ratio is relatively small as compared to the other programs because almost all the keys need to be transferred between the processors and this could lead to a congested network, leading to slow execution of the program. A key issue in the performance of radix sort is the choice of the radix.

The number of operations required for sorting ***n*** numbers with ***k*** digits and radix ***s*** in each number would be ***2nk+2ks***[Radix03]. Thus, for 4096 keys, the number of operations required would be 369264.

6.3 Median Filter and Sobel's Edge detection

The speedup obtained for the Median filter implementation on the SCMP was very close to the ideal speedup. This is because of the fact that there is very little communication required, and the calculations done after the communication step are not dependent on results of other processors. Every processor requests the top and the bottom

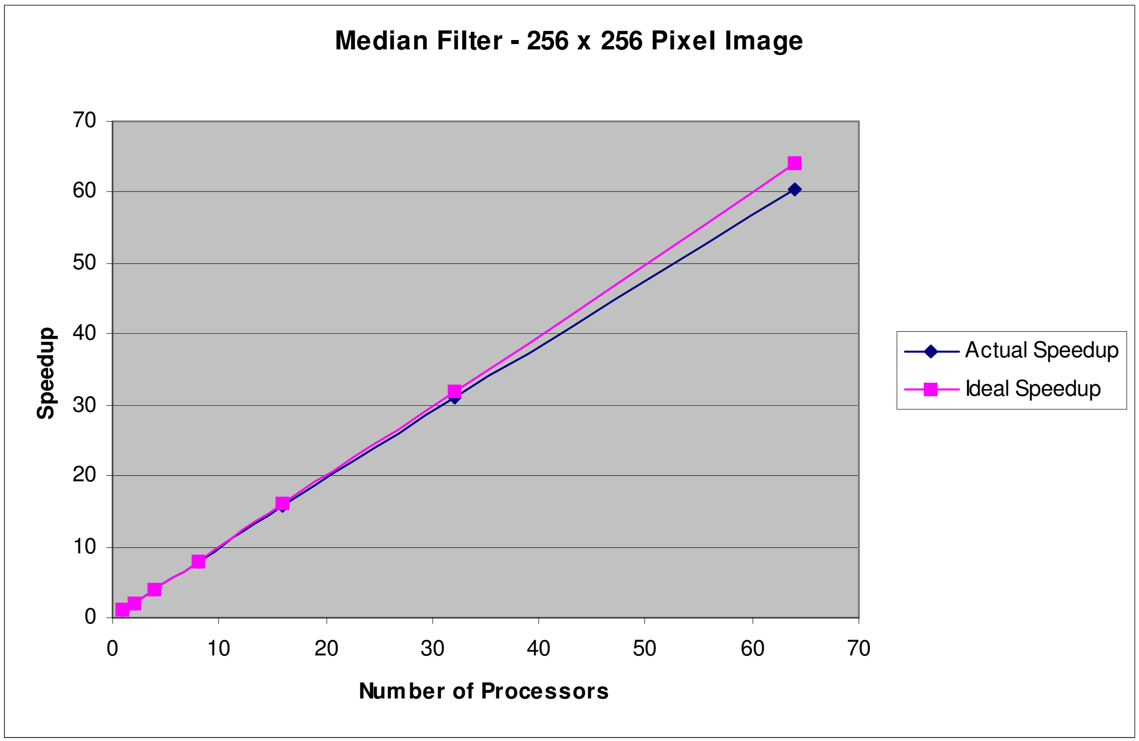


Figure 6.4: *Median Filter Speedup*

row of the image it owns from the processor right above and below it. At this point, it has all the information it requires to apply the median filter to the image it owns. After the calculations are done, every processor writes it part of the image to the output file. The speedup graph for the Median filter algorithm is shown in Figure 6.4. Similar results were observed for Sobel's edge detection algorithm, which are shown in Figure 6.5

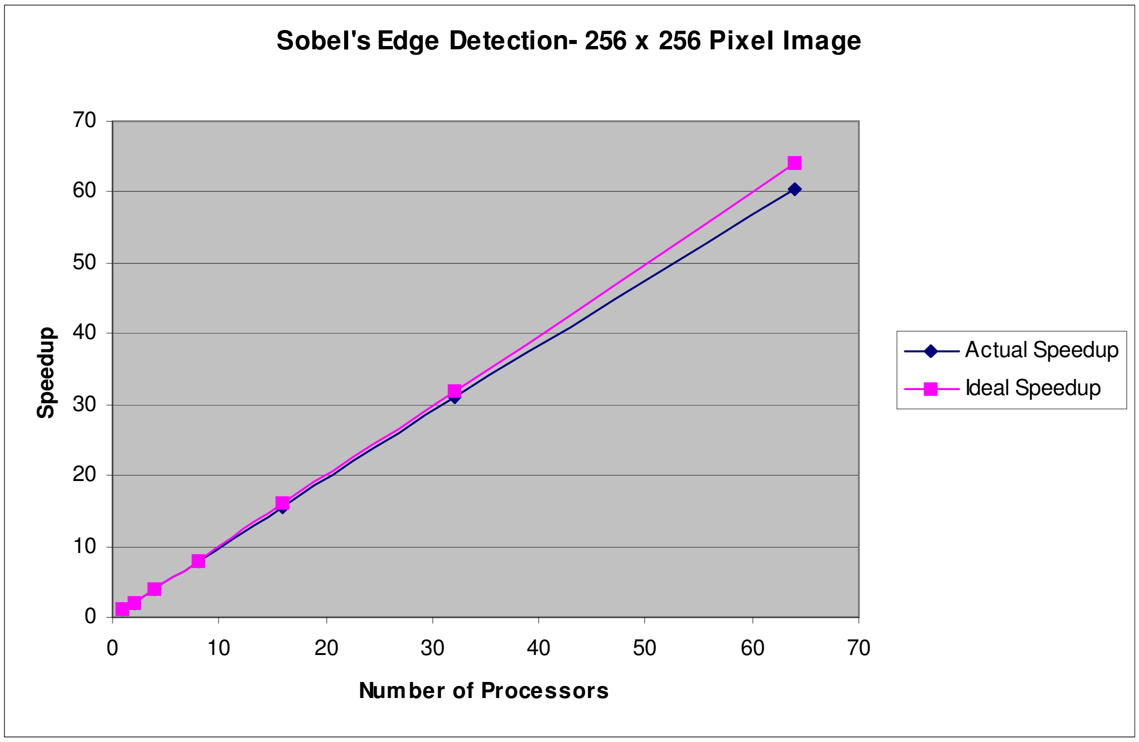


Figure 6.5: *Sobel's Edge detection Speedup*

Since the calculations are performed on a 3x3 block of an image, the average number of operations required to calculate the median for a pixel is 12. Thus, for an nxn pixel image, the total number of operations required median filter is $\boldsymbol{12n^2}$. Applying this formula for a 256-by-256 image, the number of operations required to calculate the median filter would be 786432. The comparison between a Pentium 4 2.5GHz processor and the SCMP is shows in Figure 6.6

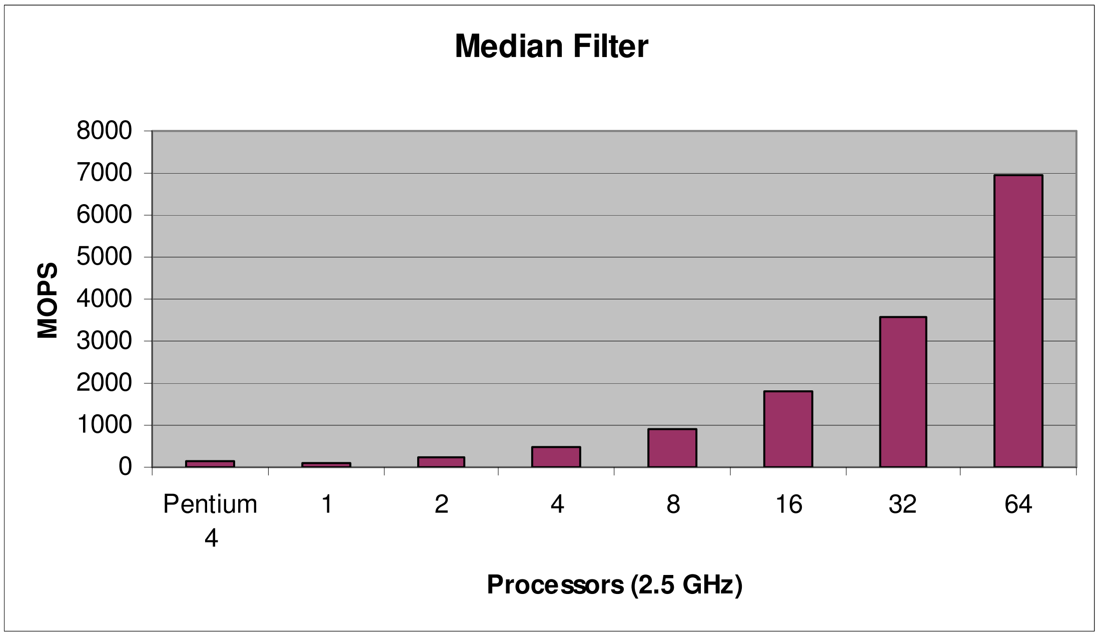


Figure 6.6: *Performance of SCMP vs. Intel Pentium 4 for the Median Filter*

From the graph we see that as the number of processors is increased, there is a significant increase in the MOPS ratings. This shows that the SCMP can successfully exploits the available thread level parallelism from the median filter and Sobel's edge detection algorithm.

6.4 LU Factorization

The graph for the speedup obtained after implementing the parallel LU factorization is shown in Figure 6.7. The graph is a for a 2048-by-2048 matrix with 16-by-16 block sizes. It was observed that the speedup for a large matrix was more than a smaller matrix. This is mainly due to the fact that the computation-to-communication ratio for larger matrices is more than the smaller matrix.

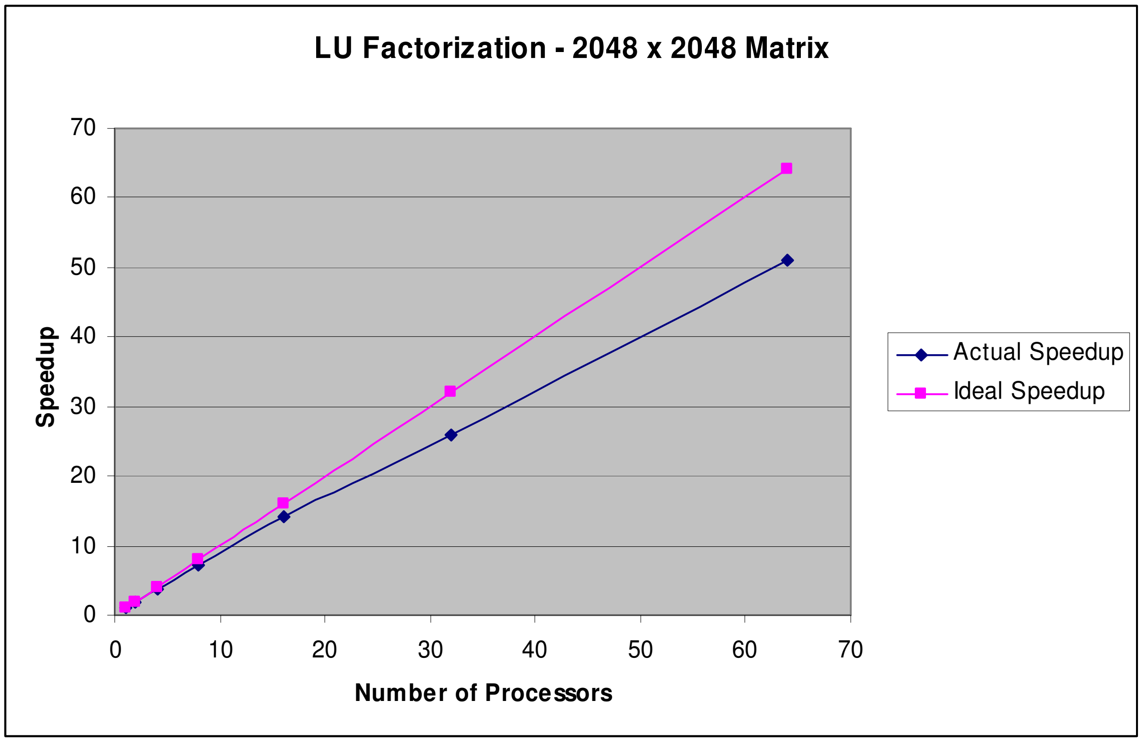


***Figure 6.7** – LU Factorization speedup for a 2048x2048 Matrix*

The reason for sub-linear speedup for the LU factorization has to do with the size of input data-sets rather than the inherent nature of the applications. A dataset of 256 x 256 keys results in a low computation to communication ratio, despite the block-oriented decompositions. Larger data sets provide more data the processor can compute on, and less wait time for data. During every computation of the pivot block, there are three barriers that the processors have to encounter. Firstly the pivot block is factorized, then the perimeter rows and columns are modified and then all the blocks in the inner matrix are updated. The pivot block has to be sent to all the perimeter blocks and the inner matrix, which results in a larger communication overhead. The overhead penalty is greatly increased while updating the inner matrix elements because every block needs two other blocks from different processors to successfully update it. Thus, the processor that owns the inner block creates threads on two other processors requesting them to send the blocks it requires for the calculation.

The total number of calculations required for the LU factorization for a n-by-n matrix is $\mathbf{2}\boldsymbol{n}^{\mathbf{3}}\mathbf{/3}$[Singh95]. The MOPS ratings for the SCMP is compared with the sequential P4 2.5GHz processor implementation.

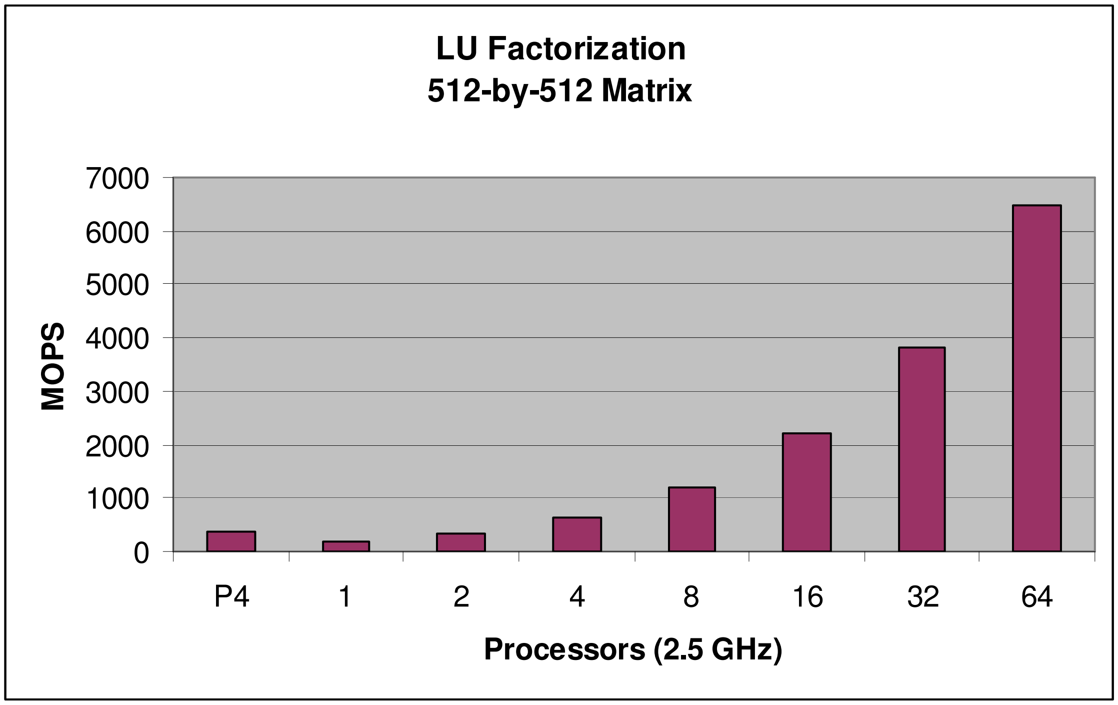


Figure 6.8 – *Performance of SCMP vs. Intel Pentium 4 for LU 512-512 matrix*

From the graphs in Figure 6.8 and Figure 6.9 we can see that the performance of a Pentium 4 machine is similar to 2 SCMP processors. Thus, for the Pentium 4 system, which aggressively tries to exploit ILP, we get a speedup of only about 2 over a single SCMP processor. The SCMP does not exploit any ILP for the algorithm which leads to the conclusion that there is only a certain amount of ILP that could be exploited for the LU algorithm. However, we get a good speedup as we increase the number of SCMP processors, which exploits thread-level parallelism. Thus, as we increase the number of processors, we get a higher MOPS rating.

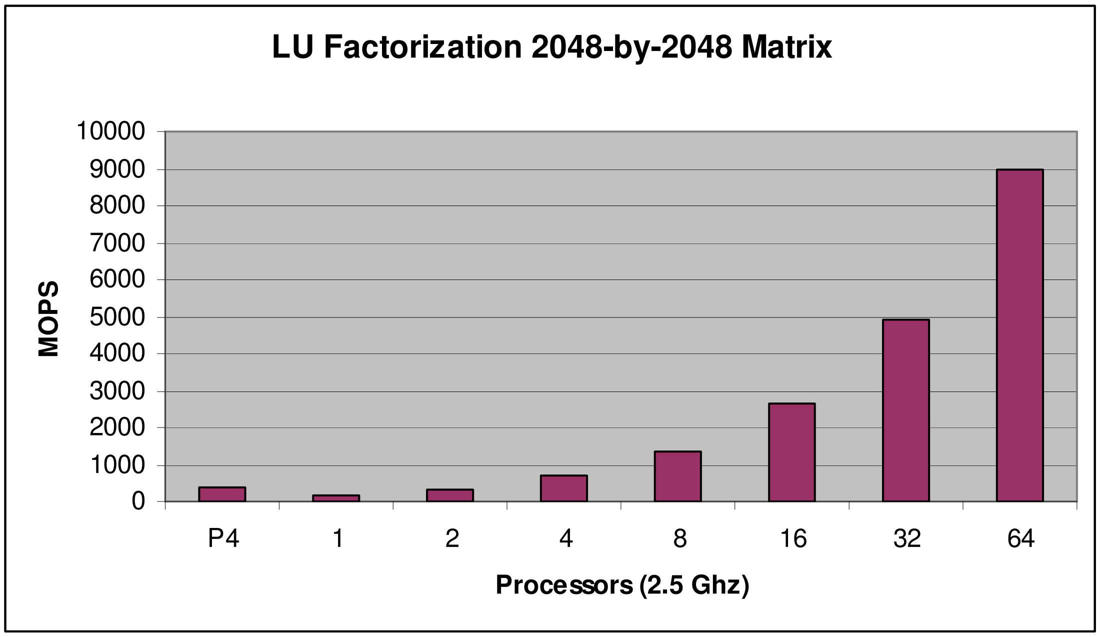


***Figure 6.9 -** Performance of SCMP vs. Intel Pentium 4 for LU 2048-2048 matrix*

7. Conclusions

Existing benchmarks don't accurately measure the performance of the SCMP due to the fact that the benchmarks were originally designed for a specific system, which could be very different from the SCMP. Every computer has its own set of instructions and is developed for a specific purpose. Evaluating the performance of a parallel computer is a function of various issues such as the architecture, memory management, I/O schemes and its instruction set. This thesis has attempted to demonstrate how the SCMP could be used to write a broad range of application programs. The applications developed were designed to exploit the unique features of the SCMP. The SCMP system could be used in bioinformatics where the Smith-Waterman algorithm is implemented in parallel. Image processing algorithms such as the Median filter and Sobel's edge detection were also implemented and significant speedup was observed. Radix sort and LU factorization were the kernel benchmarks, which were designed from SPLASH-2 benchmarks. The benchmarks designed have been written to exploit the message-passing architecture of the SCMP with communication occurring through data and thread driven messages.

Future work on the SCMP application development could include more full-blown industry standard benchmarks. The Fast Fourier Benchmark (FFW – www.fftw.org) is an industry standard computing the discrete Fourier transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size.

8. References

[Agarwal97] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring It All to Software: Raw Machines," *Computer*, vol. 30, no. 9, September 1997, pp. 86-93.

[Agarwal00] V. Agarwal et al., "Clock Rate versus IPC: The End of the Road for Conventional Microprocessors." Proc. 27th Ann. Int'l Symp. Computer Architecture, New York: ACM Press, 2000, pp. 248-259.

[Athas88] W. C. Athas and C. L. Seitz, "Multicomputers: Message-Passing Concurrent Computers." IEEE Computer, vol. 21, no. 8, Aug. 1988, pp. 9-24.

[Baker02] J. M. Baker et al., "SCMP: A Single-Chip Message Passing Parallel Computer." Proc. Parallel and Distributed Processing Techniques and Applications, PDPTA'02, CSREA Press, 2002, pp. 1485-1491.

[Baker03] J. M. Baker et al., "SCMP: A Single-Chip Message Passing Parallel Computer." Journal of Supercomputing, '03

[Culler98] D. E. Culler and J. P. Singh, with A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, San Francisco: Morgan-Kaufmann, 1998.

[Dubey97] K. Diefendorff and P. Dubey, "How Multimedia Workloads Will Change Processor Design," *Computer*, vol. 30, no. 9, September 1997, pp. 43-45.

[Dally99] W.J. Dally and S. Lacy, "VLSI Architecture: Past, Present, and Future*," 20th Conference on Advanced Research in VLSI (ARVLSI 99)*, March 1999.

[Ghosh99] P. Ghosh, R. Mangaser, C. Mark, and K. Rose, "Interconnect-Dominated VLSI Design," *20th Conference on Advanced Research in VLSI (ARVLSI 99)*, March 1999.

[Matzke97] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?," *Computer*, vol. 30, no. 9, September 1997, pp. 37-39.

[Diefendorff99] K. Diefendorff, "Power4 Focuses on Memory Bandwidth," *Microprocessor Report*, vol. 13, no.13, October 6, 1999.

[Tremblay2000] M. Tremblay, J. Chan, S. Chaudhry, A.W. Conigliaro, and S.S. Tse, .The MAJC Architecture: A Synthesis of Parallelism and Scalability,. *IEEE Micro*, vol. 20, no. 6, November-December 2000, pp. 12- 25.

[Chang96] [15] K. Olukotun, B.A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," *Seventh International Symp. Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996, pp. 2-11.

[Allen01] F. Allen, et. al., "Blue Gene: A Vision for Protein Science Using a Petaflop Supercomputer," *IBM Systems Journal*, vol. 40, no. 2, 2001, pp. 310-327.

[Krishnan99] V. Krishnan and J. Torrellas, "A Chip-Multiprocessor Architecture with Speculative Multithreading," *IEEE Transactions on Computers*, vol. 48, no. 9, September 1999, pp. 866-880

[Peb03] Radix Sort, National Institute of Standards and Technology. Accessed August 12, 2003.
http://www.nist.gov/dads/HTML/radixsort.html

[Radix03] Parallel Radix Sort, Joint Institute of Computational Science. Accessed August 12, 2003.
http://www.jics.utk.edu/PCUE/MOD10_SSS/tsld029.htm

[Radix00] Radix sort example, Accessed August 12, 2003.
www.cag.lcs.mit.edu/6.893-f2000/ readslides/Vector_Sort_Hash.pdf

[Moore99] Ross Moore, LU Factorization Example, University of New England, Armidale, NSW, Australia. Accessed August 12, 2003
http://turing.une.edu.au/~amth247/Lectures/Lecture_08/lecture/

[Ganapati93] CoMet: A synthetic benchmark for message-passing architectures. Master's thesis, Oregon Graduate Institute of Science & Technology, July 1993

[Stewart79] J. Dongarra, J. Bunch,, C.Moler and G.W. Stewart. LINPACK User's Guide. SIAM, Philadelphia, PA 1979

[Blelloch91] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pp. 3-16, July 1991.

[Singh93] Steven Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The Performance Advantages of Integrating Message Passing in Cache-Coherent Multiprocessors. Stanford University Technical Report No. CSL-TR-93-593, December 1993.

[Singh95] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Stanford University Technical Report No. CSLTR- 95-660. January 1995.

[Hennessy02] John L. Hennessy, D. A. P., David Goldberg (2002). Computer Architecture: A Quantitative Approach.

[Berry89] M. Berry et al., The Perfect Club Benchmarks: Effective performace evaluation of supercomputers, The International Journal of Supercomputer Applications, 3 (1989), pp 5-40.

[Dongarra79] J.J. Dongarra, J. Bunch, C. Moler, and G.W.Stewart. LINPACK *User's Guide*. SIAM, Philadelphia, PA 1979

[Gray93] J. Gray, The Benchmark Handbook for Database and Transaction Processing Systems, San Franciso, CA: Morgan Kaufmann http://www.tpc.org

[Almasi89] Almasi, G. S. and Gottlieb, A. (1989). Highly Parallel Computing. Benjamin/Cummins.

Vita

Jignesh Shah was born on October 2nd, 1979 in Baroda, India. In 1997 he graduated from M.E.S Indian School in Doha, Qatar. After high school he started his Bachelor of Science in Computer Engineering at Virginia Tech, Blacksburg, VA and graduated in 2002. He continued on at Virginia Tech and plans to finish his Masters in Computer Engineering in 2004. A Graduate Research Assistantship at Virginia Bioinformatics Institute in Blacksburg, VA supported his Masters degree.

In September he is scheduled to start working at Microsoft in Redmond, WA as a Software Test Engineer in the Windows Security Team.