PARALLELIZATION AND PERFORMANCE OPTIMIZATION OF BIOINFORMATICS AND BIOMEDICAL APPLICATIONS TARGETED TO ADVANCED COMPUTER ARCHITECTURES

by

Yanwei Niu

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Summer 2005

PARALLELIZATION AND PERFORMANCE OPTIMIZATION OF BIOINFORMATICS AND BIOMEDICAL APPLICATIONS TARGETED TO ADVANCED COMPUTER ARCHITECTURES

by

Yanwei Niu

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____
Guang R. Gao, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____
Kenneth E. Barner, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____
Charles Boncelet, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: ____
Li Liao, Ph.D.
Member of dissertation committee

To My Parents

ACKNOWLEDGMENTS

I am indebted to many people for the completion of this dissertation. First and foremost, I would like to thank my two co-advisors, Professor Kenneth Barner and Professor Guang R. Gao. I thank them for the support, encouragement, and advisement which are essential for the progress I have made in the last five years. I would not have been able to complete this work without them. Their dedication to research and their remarkable professional achievements have always motivated me to do my best.

I also would like to thank other members of my advisory committee who put effort to reading and providing me with constructive comments: Professor Li Liao and Professor Charles Boncelet. Their help will always be appreciated.

My sincere thanks also go to numerous colleagues and friends at the Information Access Laboratory and the CAPSL Laboratory, including Yuzhong Shen, Beilei Wang, Lianping Ma, Bingwei Weng, Ying Liu, Weirong Zhu, and Robel Kahsay, among many others. I am especially thankful to Yuzhong Shen for his help with using Latex and numerous software tools. I am grateful to Ziang Hu, Juan del Cuvillo and Fei Chen for their help with the simulation environment.

My parents, Baofeng Niu and Chaoyun Wang, certainly deserve the most credit for my accomplishment. Their support and unwavering belief in me occupy the most important position in my life. I also want to mention my sister, Junli, who is the person I can always talk to during times of frustration or depression. My friend Weimin Yan remains a continuing support to me with his perseverance and optimistic attitude towards life. My best friend, Yujing Zeng, has made my life at Newark colorful and interesting.

I also want to extend my appreciation to every single teacher that I had in my life, especially my advisor, Professor Peiwei Huang at the Shanghai Jiao Tong University, for her help and confidence in me.

This work was supported by the National Science Foundation under the grant 9875658 and by the National Institute on Disability and Rehabilitation Research (NIDRR) under Grant H133G020103. I also wish to thank Dr. John Gardner, View Plus Technologies, for his feedback and access to the TIGER printer. Many thanks to Dr. Walid Kyriakos and his student Jennifer Fan at the Harvard Medical School for the sequential code of SPACE RIP and constructive discussions. The email suggestion about complex SVD from Dr. Zlatko Drmac at the University of Zagreb is also highly appreciated. I would like to thank the Bioinformatics and Microbial Genomics Team at the Biochemical Sciences and Engineering Division of DuPont CR&D for the useful discussions during the course of this work. I thankfully acknowledge the experiment platform support from the HPC System Group and the Laboratory Computing Resource Center at the Argonne National Laboratory.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In this dissertation, we focus on three representative applications targeted to advanced computer architectures: parallel Hmmpfam (Hidden Markov Model for Protein FAMily database search) on cluster computing, parallel SPACE RIP (Sensitivity Profiles From an Array of Coils for Encoding and Reconstruction in Parallel) on Cyclops-64, a state-of-the-art multiprocessor-on-a-chip computer architecture, and halftoning-based tactile graphics.

Hmmpfam is one of the widely used bioinformatics tools for searching a single sequence against a protein family database. We analyzed the Hmmpfam program structure, proposed a new task decomposition scheme to reduce data communication and implemented a scalable and robust cluster-based parallel Hmmpfam using the EARTH (Efficient Architecture for Running Threads) model.

SPACE RIP, one of the parallel imaging techniques, utilizes a number of receiver coils to simultaneously acquire data, thus reducing the acquisition time. We implemented the parallelization and optimization of SPACE RIP at three levels. The top level is the loop level parallelization, which decomposes SPACE RIP into many tasks of a singular value decomposition (SVD) problem. The middle level parallelizes the SVD problem using the one-sided Jacobi algorithm and is implemented on Cyclops-64. At this level, an SVD problem is decomposed into many matrix column rotation routines. The bottom level further optimizes the matrix column rotation routine using several memory preloading or loop unrolling approaches. We developed a performance model for the dissection of total execution cycles into four parts and used this model to compare different memory access approaches.

We introduced halftoning algorithms into the field of tactile imaging and implemented four different multilevel halftoning algorithms in the TIGER (Tactile Graphics Embosser) printer, a widely used embossing printer designed to produce tactile text and graphics for visually impaired individuals. Digital halftoning creates the illusion of a continuous-tone image from the judicious arrangement of binary picture elements. We exploited the TIGER tactile printer's variable-height punching ability to convert graphics to multilevel halftoning tactile texture patterns. We conducted experiments to compare the halftoning-based approach with the simple, commonly utilized thresholding-based approach and observed that the halftoning-based approach achieves significant improvement in terms of its texture pattern discrimination ability.

Chapter 1

INTRODUCTION

1.1 Background and Motivation

Over the past few years, many new design concepts and implementations of advanced computer architectures have been proposed. Among several new trends, building clustering servers for high performance computing is gaining more acceptance. Assembling large Beowulf clusters [1] is easier than ever, and their performance is increasing dramatically. In TOP500 [2], a website founded in June 1993 which assembles and maintains a list of the 500 most powerful computer systems in the world, there are 294 clusters in the November 2004's list, up from 208 in November 2003 and 94 in November 2002.

Another important trend is the emerging multithreaded architectures [3–8]. Traditional approaches to boosting CPU performance, such as increasing clock frequency, execution optimization , as well as increasing the size of cache [9] are running into some fundamental physical barriers. Multithreaded architectures have the potential to push the computer architecture paradigm to a new limit by exploring thread-level parallelism. Meanwhile, technology development will produce chips with billions of transistors, enabling large quantities of logic and memory to be placed on a single chip. One chip can have many thread units with independent program counters. The IBM Cyclops-64 [10–12] architecture is an example of multithreaded architectures.

In this dissertation, we focus on three bioinformatics or biomedical related applications and conduct parallelization and performance optimization targeted to the emerging computer architectures. Bioinformatics and biomedical applications provide both challenges and opportunities for parallel computing. For instance, the genome projects and many other sequencing projects generate a huge amount of data. Comprehension of those data and the related biological processes becomes almost impossible without harnessing the power of parallel computing and advanced computer architectures. Examples of highly computation-intensive applications include database searching [13, 14], protein folding [15], phylogenetic tree reconstruction [16], etc. In the biomedical imaging field, applications such as image reconstruction [17], image registration [18, 19] and fMRI image sequence processing [20] are a few representative examples.

In the remainder of this chapter, we explore a few general concepts about computer architectures and provide background information about bioinformatics. We then summarize our major contributions and publications.

1.2 Parallel Computing Paradigms

Parallel computing or parallel processing is the solution of a single problem by simultaneous execution of different parts of the same task on multiple processors [21]. The terms “High Performance Computing (HPC)”, “parallel processing” and “supercomputing” are often used interchangeably.

According to Flynn’s taxonomy of computer architecture [22], parallel computing architectures are divided into two large classes: Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) machines. In SIMD architectures, once each of multiple processing elements (PEs) is loaded with data, a single instruction from the central processing unit (CPU) causes each of the PEs to perform the indicated instruction at the same time, as in a vector processor or array processor. An example of SIMD applications is in the area of image processing: changing the brightness of an image involves simultaneously changing the R, G, and B values of each pixel in the image. In MIMD architectures, there are multiple processors each dealing with their own data. Examples include a multiprocessor, or a network of workstations. The MIMD architectures can be classified according to their programming models as either shared memory or distributed memory architectures, shown in Fig. 1.1.

1.2.1 Shared Memory Architectures

In shared memory architectures, multiple processors are connected to a global memory system including multiple memory modules, such that each processor can access any memory module [23]. Most commonly, a single address space is employed in a shared memory architecture, which means that each location in the global memory system is associated with a unique address. This address is used by each processor to access the location.

The shared memory architecutures can be further classified as either Non Uniform Memory Access (NUMA) or Uniform Memory Access (UMA) models. In the NUMA model, the access time to the shared memory varies with the location of the processor. In the UMA model, all processors have equal access time to the whole memory which is uniformly shared by all processors.

Symmetric Multiprocessor (SMP) is a shared memory multiprocessor where the cost of accessing a memory location is the same for all processors. Software for SMP machines is usually custom programmed for multithreaded processing. However, most consumer products such as word processors and computer games are not written in such a manner because writing a program to increase performance on SMP systems will produce a performance loss on uniprocessor systems, which comprise the largest percentage of the market. Therefore, these products cannot gain large benefits from SMP systems.

The advantage of shared memory architectures is that they are relatively easy to write software for due to the convenience of sharing data. However, due to the single address space concept, variable sharing may limit the speedup of the computation. Locks and semaphores used to avoid memory conflicts are also very costly. In the shared memory systems, many CPUs need fast access to memory and will likely cache memory. A cache architecture with a strong consistency model is not scalable.

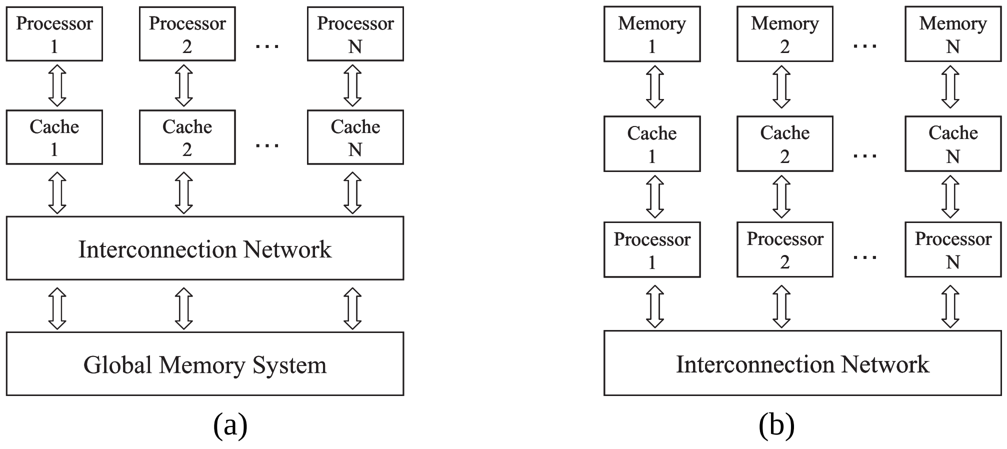


Figure 1.1: (a)Shared memory architure (b) Distributed memory architure

Examples of shared memory architectures include SUN Sunfire SMP, Cray T3E, Convex 2000 and SGI Origin/Onyx [24]. Entry level servers and workstations with two processors dominate the SMP market today; mid level servers usually have four to eight processors. At the high end, the fastest single SMP system is the 504 processor Cray X1 at the Oak Ridge National Laboratory, which is ranked number twenty nine on the list of the world's Top 500 Supercomputers [2] as of November 2004.

1.2.2 Distributed Memory Architectures

In distributed memory MIMD architectures, the memory is associated with individual processors and a processor is only able to address its own memory, as shown in Fig. 1.1b. Since these systems lack shared memory, data is communicated by message-passing via the interconnection network. Thus such multiprocessor systems are usually called message-passing multiprocessors.

The advantage of distributed memory architectures is that they can physically scale easier than shared memory multiprocessors. Scaling up distributed memory machines is simply adding communication links to connect additional processors to existing processors. The drawback is that its message-passing mechanism is not as attractive for programmers. It usually requires the programmers to provide the explicit message-passing calls in the code. This may be problematic for applications that require sharing large amounts of data.

Intel Paragon, CM-5, and Transputers [24] are a few examples of distributed memory machines. Cluster computing and grid computing are two of the most popular examples.

1.2.2.1 Cluster Computing

A computer cluster is a group of independent computers connected into a unified system through software and networking [21]. One of the most popular implementations is a cluster-based on commodity hardware, on a private system network, with an open source software (Linux) infrastructure. This configuration is often referred to as a Beowulf cluster [1]. The Beowulf Project was started in early 1994. The initial prototype was a cluster computer consisting of 16 DX4 processors connected by a channel bonded Ethernet. The top supercomputer as of November 2004 is the Department of Energy's BlueGene/L cluster system [25].

There are several factors that have contributed to the success of the Beowulf cluster project. First of all, market competition has driven the prices down and reliability up for the subsystems, including microprocessors, motherboards, disks and network systems. Secondly, open source software, particularly the Linux operating system, GNU compilers, programming tools, MPI and PVM message-passing libraries are now available. Thirdly, an increased reliance on computational science demands high performance computing. Typical applications include bioinformatics, financial market modelling, data mining, and Internet servers for audio and games.

PVM (Parallel Virtual Machine) [26] and MPI (Message Passing Interface) [27] are software packages for parallel programming on a cluster. PVM used to be the standard until MPI appeared. PVM was developed by the University of Tennessee, the Oak Ridge National Laboratory and Emory University. The first version was written at ORNL in 1989. MPI is the standard for portable message-passing parallel programs. It is a library of routines that can be called from Fortran, C and C++ programs. MPI's advantage over older message-passing libraries is that it is both portable and fast.

1.2.2.2 Grid Computing

CERN (an European Organization for Nuclear Research), which was a key in the creation of the World Wide Web, defines the "Grid" as: "a service for sharing computer power and data storage capacity over the Internet" [21]. Grid computing offers a model for solving massive computational problems by making use of the unused resources of large numbers of computers treated as a virtual cluster embedded in a distributed telecommunications infrastructure.

Grid computing has the following features: (1), It allows the virtualization of disparate IT resources. (2), It allows the sharing of resources, which include not only files but also computing power. (3), It is often geographically distributed and heterogeneous, which makes it different from cluster computing.

Typical applications of grid computing include grand challenging problems like protein folding, financial modeling, earthquake simulation, climate/weather modelling etc. An example of grid computing is BIRN (Biomedical Informatics Research Network), which is a National Institutes of Health initiative providing an information technology infrastructure, notably a grid of supercomputers, for distributed collaborations in biomedical science.

1.2.3 Multithreaded Architectures

The design concept of computer architecture over the last two decades has been mainly on the exploitation of the instruction level parallelism, such as pipelining, VLIW

(Very Long Instruction Word) or superscalar architecture [24]. Pipelining is now universally implemented in high-performance processors. Superscalar means the ability to fetch, issue to execution units, and complete more than one instruction at a time. Similar to superscalar architectures, VLIW enables the CPU to execute several instructions at the same time and uses software to decide which operations can run in parallel. Superscalar CPUs, in contrast, use hardware to decide which operations can run in parallel.

However, the major processor manufactures have run out of room with the traditional approaches to boosting CPU performance, such as increasing clock frequency, execution optimization (pipelining, branch prediction, VLIW and superscalar), as well as increasing the size of cache [9]. First of all, as the clock frequency increases, the transistor leakage current also increases, leading to excessive power consumption. Second, the design concepts of traditional approaches have become too complex. Third, resistance capacitance delays in signal transmission grow as feature sizes shrink, imposing an additional bottleneck that frequency increases do not address. Also, for certain applications, traditional serial architectures are becoming less efficient as processors get faster due to the effect of the Von Neuman bottleneck [28].

In addition, the advantages of higher clock speeds are negated by memory latency. There are many commercial or scientific applications which have frequent memory access, and the performance of such applications is dominated by the cost of memory access. As pointed out in many papers, microprocessor performance has been doubling every 18-24 months for many years, while DRAM performance only improves by 7% per year [29]. Therefore the memory access latency continues to grow in terms of CPU cycles. The divergence of the CPU and memory speed is generally referred to as the “memory wall” problem. Assuming 20% of the total instructions in a program need to access memory, which means, one of every five instructions during execution accesses memory, the system will hit the memory wall if the average memory access time is greater than 5 instruction cycles [30].

Therefore, for the next generation of computer architectures, multithreaded architectures are becoming more popular. Depending on the specific form of a multithreaded processor, a thread could be a full program (single-threaded UNIX process), an operating system thread (a light-weighted process, e.g., a POSIX thread [31]), a compiler-generated thread, or a hardware generated thread [5].

1.2.3.1 Classification

Multithreaded models can be classified according to the thread scheduling mechanisms (preemptive or non-preemptive), architectural features and memory models (shared memory or distributed memory), or program flow mechanisms (dataflow or control flow) [32].

According to the classification in [5], multithreaded architectures in the more narrow sense only include architectures that stem from the modification of scalar RISC, VLIW, or superscalar processors. The classification from [5] is shown in Fig. 1.2. The terminologies in the figure are explained as follows:

- Threaded dataflow [33]: a combination of multithreading and the dataflow model. This model uses the dataflow principle to start the execution of a non-preemptive thread.
- Explicit multithreading [5]: an approach that explicitly executes instructions of different user-defined threads (operating system threads or processes) in the same pipeline.
- Implicit multithreading [5]: an approach that adopts thread level speculation, dynamically generates speculative threads from single-threaded programs and executes them concurrently with the lead thread.

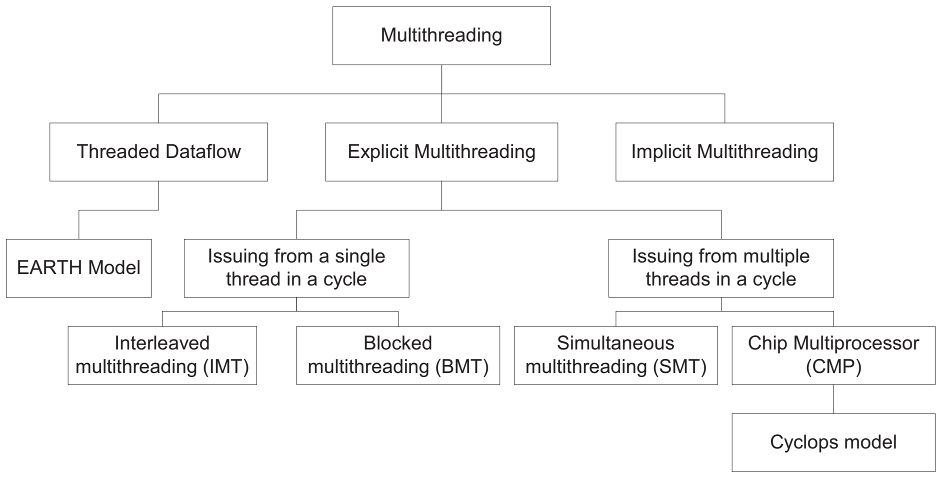

Multithreading
Threaded Dataflow
Explicit Multithreading
Implicit Multithreading
EARTH Model
Issuing from a single thread in a cycle
Issuing from multiple threads in a cycle
Interleaved multithreading (IMT)
Blocked multithreading (BMT)
Simultaneous multithreading (SMT)
Chip Multiprocessor (CMP)
Cyclops model

Figure 1.2: Classification of multithreaded architecture

- Interleaved multithreading (IMT) [24]: an approach in which an instruction of different threads is fetched and fed into the execution pipeline at each processor cycle. The Cray MTA chip is a VLIW pipelined processor using the IMT technique.

- Blocked multithreading (BMT) [24]: an approach in which the instructions of a thread are executed successively until an event occurs that may cause latency and induces a context switch.

- Simultaneous multithreading (SMT) [7]: an approach that simultaneously issues instructions from multiple threads to the execution units of a superscalar processor.

- Chip MultiProcessor (CMP) [10–12, 34, 35] : a single chip design that uses a collection of independent processors with less resource sharing. This approach may also be referred to as "multiprocessor-on-a-chip" or "multicore processor" design.

In this section, we first introduce the dataflow model and multithreading, followed by a brief introduction of CMP and the Cyclops-64 architecture.

1.2.3.2 Data Flow Multithreading

The dataflow model [6, 36] is a dramatic break from the traditional von Neumann model [37]. In the von Neumann computer, a single program counter determines which instruction to execute next and a complete order exists between instructions. The dataflow model, in contrast, only has a *partial* order between instructions. The fundamental idea of dataflow is that any instruction can be executed as long as its operands are present.

The combination of the von Neumann model and the dataflow model [38] puts two or more dataflow actors into threads; therefore, it can reduce fine-grain synchronization costs and improve locality in dataflow architectures. It can also add latency-tolerance and efficient synchronization to conventional multithreaded machines by integrating dataflow synchronization into the thread model.

According to Dennis and Gao [33], a thread is viewed as a sequentially ordered block of instructions with a grain-size greater than one. Evaluation of a non-preemptive thread starts as soon as all input operands are available, adopting the idea of the dataflow model. Access to remote data is organized in a split-phase manner by one thread sending the access request to memory and another thread activating when its data is available. Thus a program is compiled into many, very small threads activating each other when data become available.

EARTH (Efficient Architecture for Running THreads) [3, 4] is one example of the multithreaded data flow model. More details of the EARTH model are presented in next chapter. The EARTH model is currently supported on both SMP machines and cluster computers.

1.2.3.3 Multiprocessor-on-a-Chip Model

The “multiprocessor-on-a-chip” model is a single chip design that uses a collection of independent processors with less resource sharing. Currently available multicore processors include IBM’s dual-core Power4 and Power5, Hewlett-Packard’s PA-8800 and

Sun’s dual-core Sparc IV. AMD will deliver dual-core Opterons around the middle of 2005. Intel also makes shifts to multicore chips this year.

Intel researchers and scientists are experimenting with many tens of cores, potentially even hundreds of cores per die. And those cores will support tens, hundreds, maybe even thousands of simultaneous threads [39]. Intel’s Platform 2015 [28] describes the evolution of its multiprocessor architecture over the next 10 years. Multicore architectures of Platform 2015 and sooner will enable dramatic performance scaling and address important power management and heat challenges. They will be able to activate only the cores needed for a given function and power down the idle cores. The features of the hypothetical Micro 2015 include: (1), Parallelism will be handled by an abundant number of software and hardware threads. (2), A relatively large high speed, reconfigurable onchip memory will be shared by groups of cores, the OS, the micro kernel and the special-purpose hardware. (3), Tera-flops of supercomputer-like performance and new capabilities for new applications and workloads will be provided.

Another representative “multiprocessor-on-a-chip” architecture design is Cyclops-64 [10–12, 34, 35], a new architecture for high performance parallel computers being developed at the IBM T. J. Watson Research Center and the University of Delaware. The basic cell of this architecture is a single chip with multiple threads of execution, embedded memory, and integrated communication hardware. The memory latency is tolerated by the massive intra-chip parallelism. More details of Cyclops-64 architecture are described in Section 3.2.

1.3 Bioinformatics

1.3.1 Definition

Bioinformatics or computational biology is the application of computational tools and techniques to the management and analysis of biological data [40]. It is a rapidly evolving discipline and involves techniques from applied mathematics, statistics, and computer science. The terms "bioinformatics" and "computational biology" are often used interchangeably, although the latter typically focuses on algorithm development and specific computational methods.

We view bioinformatics research as an integration of biological data management and knowledge discovery. Biological data management enables efficient storage, organization, retrieving, and sharing of different types of information. Knowledge discovery involves the development of new algorithms to analyze and interpret various types of data, as well as the development and implementation of software tools.

Bioinformatics has many practical applications in different areas of biology and medicine. More specifically, major research efforts in the field include sequence alignment, gene finding, genome assembly, protein structure alignment and prediction, and the modeling of evolution.

1.3.2 Role of High Performance Computing

Bioinformatics has inspired computer science advances with new concepts, new ideas and new designs. In turn, the advances in computer hardware and software algorithms have also revolutionized the area of bioinformatics. The cross-fertilization has benefited both fields and will continue to do so. The role of high performance computing in bioinformatics can be reflected from two angles: (1) large amounts of data; (2) computationally challenging problems.

Firstly, large amounts of data create an urgent need for high performance computing. For example, the genetic sequence information in the National Center for Biotechnology GenBank (NCBI) database [41] has more than 44 billion base pairs as of April

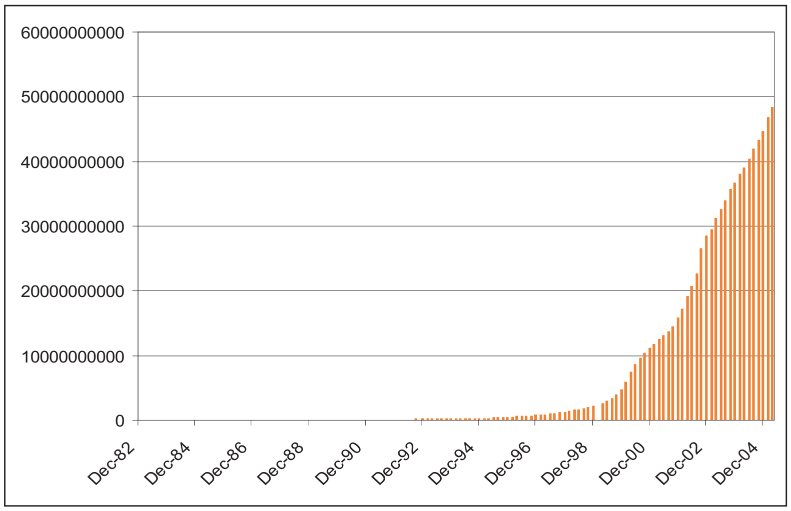


Figure 1.3: NCBI database growth (Number of base pairs)

2005. The growth of the NCBI database is shown in Fig. 1.3. A base pair is a pair of nitrogenous bases held together by hydrogen bonds that form the core of DNA and RNA, i.e the A:T, G:C and A:U interactions.

In bioinformatics, BLAST (Basic Local Alignment Search Tool) [42] is an algorithm for comparing biological sequences, such as the amino-acid sequences or the DNA sequences. Given a library or database of sequences, a BLAST search enables a researcher to look for sequences that resemble a given sequence of interest. A study [43] of the performance of BLAST found that a query on the 2003 GenBank data using a 2003 Intel-based server takes an average of around 260 seconds. The same task took only 83 seconds on a 2001 GenBank collection and 2001 hardware, and 36 seconds for 1999. BLAST is becoming around 64 percent slower each year despite improvements in hardware.

Secondly, many computational-intensive algorithms exists in the bioinformatics area. Of them there are two grand challenges: understanding evolution and the basic structure and function of proteins. A phylogenetic tree is a tree reconstructed from DNA or protein sequences to represent the history of evolution. Phylogenetic tree reconstructions involve solving difficult optimization problems with a complexity of $(2n-5)!!$ for a tree with n leafs and requires months to years of computation. Many approaches have been proposed to reconstruct phylogenetic tree using the power of high performance computing, such as parallel fast DNAml [44] and GRAPPA [16, 45]. These approaches still have limitations such as tree size and accuracy. The protein folding simulation is a popular way to predict the structure and function of proteins. Protein folding refers to a process by which a protein assumes its three-dimensional shape with which they are able to perform their biological function. According to an estimate [46], accurate simulation of a protein folding to predict the protein 3D structure may be intractable without PetaFLOPS-class computers. Simulating 100 microseconds of protein folding would take three years on even a PetaFLOPS system or keep a 3.2GHz microprocessor busy for the next million centuries. Therefore, new approaches of high performance computing and algorithmic design need to be developed to meet these challenges.

1.4 Achievements and Contributions

The principal goal of this research is to find better solutions for important and practical bioinformatics or biomedical applications. The contribution of our work is two-fold. On the one hand, we parallelize and optimize the real, large scale applications, dramatically decreasing the time for computation. On the other hand, we experimental results of the applications provide new insights for the design of computer architectures.

The applications include the Hmmpfam database search program from the bioinformatics area, the SPACE RIP image reconstruction from the biomedical area, and another very useful biomedical application – using halftoning algorithms to make graphics available to people with visual impairment. These three applications fall into the general umbrella of bio-oriented applications. The first two applications are closely related to parallel computing; the last application, in contrast, is not as closely related. This is, however, a natural result of the multi-disciplinary characteristic of this research.

Hmmpfam is a widely-used computation-intensive bioinformatics software for sequence classification. Sequence classification plays an important role in bioinformatics to predict the protein structure and function. The major achievements are listed as follows:

1. We analyzed the Hmmpfam program structure and proposed a new task decomposition scheme to reduce data communication and improve program scalability.

2. We implemented a scalable and robust cluster-based parallel Hmmpfam using EARTH (Efficient Architecture for Running Threads), an event-driven fine-grain multithreaded programming execution model.

3. Our new parallel scheme and implementation achieved notable improvements in terms of program scalability. We conducted experiments on two advanced supercomputing clusters at the Argonne National Laboratory (ANL) and achieved an absolute speedup of 222.8 on 128 dual-CPU nodes for a representative data set.

The application SPACE RIP (Sensitivity Profiles From an Array of Coils for Encoding and Reconstruction in Parallel) is one of the parallel imaging methods that has the potential to revolutionize the field of fast MR imaging. The image reconstruction problem of SPACE RIP is a computation-intensive task, and thus a potential application for parallel computing. The major contributions of our work are summarized as follows:

1. We implemented the parallelization and optimization of SPACE RIP at three levels. The top level is the loop level parallelization, which decomposes SPACE RIP into many tasks of a singular value decomposition (SVD) problem. The middle level parallelizes the SVD problem using the one-sided Jacobi algorithm and is implemented on Cyclops-64. At this level, an SVD problem is decomposed into many matrix column rotation routines. The bottom level further optimizes the matrix column rotation routine using several memory preloading or loop unrolling approaches.

2. We developed a model and trace analyzer to decompose the total execution cycles into four parts: total instruction counts, “DLL”, “DLF” and “DLI”, where “DLL” represents the cycles spent on memory access, “DLF” represents the latency cycles related to floating point operations, and “DLI” represents the latency cycles related to integer operations. This simple model allows us to study the application performance tradeoff for different algorithms.

3. Using a few application parameters such as matrix size, group size, and architectural parameters such as onchip and offchip latency, we developed analytical equations for comparing different memory access approaches such as preloading and loop unrolling. We used a cycle accurate simulator to validate the analysis and compare the effect of different approaches on the “DLL” part and total execution cycles.

The application of using halftoning to generate tactile graphics uses signal processing algorithms and computer technologies to aid blind people. Tactile imaging is an algorithmic process that converts a visual image into an image perceivable by the sense of touch. Tactile imaging translates a visual image into an image perceivable by the sense of touch. Digital halftoning creates the illusion of a continuous-tone image from the judicious arrangement of binary picture elements. As an extension of binary halftoning, multilevel halftoning techniques are adopted on printers that can generate multiple output levels. We exploited the TIGER (Tactile Graphics Embosser) printer's variable-height punching ability to convert graphics to multilevel halftoning tactile texture patterns. The major contributions are summarized as follows:

1. We introduced digital halftoning into the field of tactile imaging and implemented four different halftoning algorithms into the TIGER printer driver.

2. According to the specifics of the TIGER printer, we extended traditional binary halftoning algorithms to multilevel algorithms.

3. We conducted experiments to compare the halftoning-based approach with the simple, commonly utilized thresholding-based approach and observed that the halftoning-based approach achieved significant improvement in terms of its texture pattern discrimination ability.

1.5 Publications

This dissertation is based on several published works. The work on the parallel Hmmpfam is published in Cluster2003 and the International Journal of High Performance Computing and Networking (IJHPCN) [47]. The work on SPACE RIP and SVD is included in the proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems [48]. The work on tactile graphics using halftoning is summarized in a paper submitted to the IEEE Transactions on Neural Systems and Rehabilitation Engineering [49].

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapters 2, 4 and 3 focus on the applications Parallel Hmmpfam, tactile graphics, parallel SPACE RIP respectively. Chapter 2 includes background information about the Hidden Markov Model and its application in the bioinformatics area, an introduction of the Hmmpfam program, the original parallel scheme in the PVM implementation, the proposed cluster-based parallel implementation and the performance results. Chapter 3 presents the background information of Cyclops-64, the parallel imaging technique SPACE RIP, the loop level and fine level parallel scheme, a performance model for different memory access schemes, and the performance results. Chapter 4 includes a brief review of tactile printing, the TIGER printer hardware and software, a discussion on the basics of halftoning techniques, and the implementation of the halftoning algorithm into the TIGER printer graphics processing pipeline, as well as the experimental design and evaluation results. Chapter 5 concludes this dissertation.

Chapter 2

A CLUSTER-BASED SOLUTION FOR HMMPFAM

2.1 Introduction

One of the fundamental problems in computational biology is the classification of proteins into functional and structural classes or families based on homology of protein sequence data. Sequence database searching and family classification are common ways to analyze the function and structure of the sequences. A “family” is a group of proteins of similar biochemical function that are more than 50% identical [50]. Sequence homology indicates a common function and common ancestry of two DNA or protein sequences.

The family classification of sequences is of particular interest to drug discovery research. For example, if an unknown sequence is identified as belonging to a certain protein family, then its structure and function can be inferred from the information of that family. Furthermore if this sequence is sampled from certain diseases X and belongs to a family F, then X can be treated using the combination of existing drugs for F [51].

Typical approaches for protein classification include pairwise sequence alignment [13, 42], consensus patterns using motifs [52] and profile hidden Markov models (profile HMMs) [53–55]. A profile HMM is a consensus HMM model built from a multiple sequence alignment of protein families. HMM is a probabilistic graphical model used very widely in speech recognition and other areas [56]. In recent years, HMM has been an important research topic in the bioinformatics area. It is applied systematically to model, align, and analyze entire protein families and the secondary structure of sequences. A consensus sequence of a family can be determined by deriving the profile HMM. Unlike

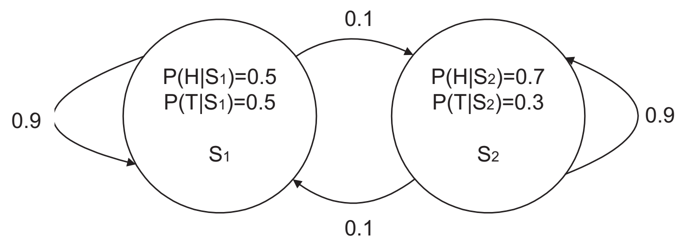


Figure 2.1: HMM model for tossing coins

conventional pairwise comparisons, a consensus model can in principle exploit additional information such as the position and identity of residues that are more or less conserved throughout the family, as well as variable insertion and deletion probabilities [57].

A very simple example of HMM for tossing coins is given in Fig. 2.1. We use a fair coin or a biased coin which has a probability of 0.7 to get a "head". We change coins with a probability of 0.1. The corresponding HMM is:

- The states are $Q = \{S_1, S_2\}$, where S_1 stands for "fair" and S_2 for "biased".
- Transition probabilities are: $\alpha_{11} = 0.9$, $\alpha_{12} = 0.1$, $\alpha_{21} = 0.1$, $\alpha_{22} = 0.9$.
- Emission probabilities are: $P(H|S_1) = 0.5$, $P(T|S_1) = 0.5$, $P(H|S_2) = 0.7$, $P(T|S_2) = 0.3$.

In this example, the observation is "Head" or "Tail". The states S_1 and S_2 are not observable, thus the name "hidden". Assuming that S_1 is the initial state, we can compute the probability of observing a certain sequence. For example:

$$P(HH|M) = P(H|S_1) \times 0.9 \times P(H|S_1) + P(H|S_1) \times 0.1 \times P(H|S_2) \tag{2.1}$$

A profile HMM can be derived from a family of proteins (or gene sequences), and later be used for searching a database for other members of the family. Fig. 2.2 is a most simplified profile model extracted from the multiple sequence alignment shown in Listing

2.1. Each block in the figure corresponds to one column in the multiple sequence alignment. The emission probabilities are listed in each block, and the transition probabilities are shown on the black arrows. The detailed process of initialization of an HMM from a multiple sequence alignment is reviewed in [57].

```
seq1: C A  − − −  A T
seq2: C A  A C T  A T
seq3: G A  C − −  A G
seq4: G A  − − −  A T
seq5: C C  G − −  A T
```

Listing 2.1: DNA sequence alignment

There are three types of questions related to profile HMM [57]: (1) How do we build an HMM to represent a family? (2) Does a sequence belong to a family? For a given sequence, what is the probability that this sequence has been produced by an HMM model? (3) Assuming that the transition and emission parameters are not known with certainty, how should their values be revised in light of the observed sequence? The problem solved in this research falls into the second category.

Usually, for a given unknown sequence, it is necessary to do a database search against an HMM profile database which contains several thousands of families. HMMER [14] is an implementation of profile HMMs for sensitive database searches. A wide collection of protein domain models have been generated by using the HMMER package. These models have largely comprised the Pfam protein family database [58–60].

Pfam (Protein families database of alignments and HMMs) is a database of protein domain families. A “domain” in the sequence context is an extended sequence pattern that indicates a common evolutionary origin. It also refers to a segment of a polypeptide chain that folds into a three-dimensional structure [50] in the “structural” context. The Pfam database contains multiple sequence alignments for each family, as well as profile HMMs for finding these domains in new sequences. Each Pfam family has two multiple

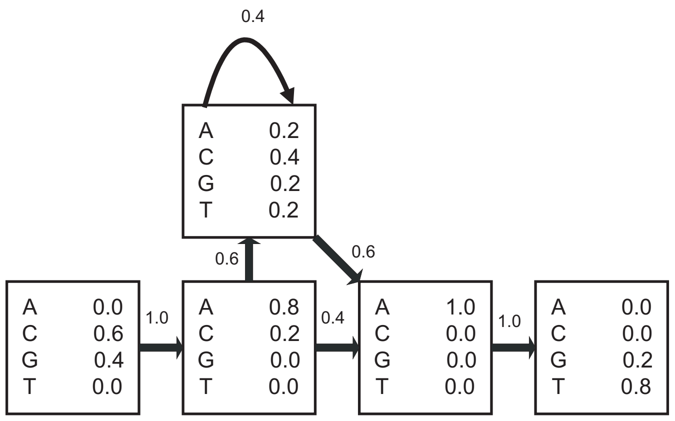


Figure 2.2: HMM model for a DNA sequence alignment

alignments: the seed alignment that contains a relatively small number of representative members of the family and the full alignment that contains all members. In the past 2 years, Pfam has split many existing families into structural domains. Currently, in the Pfam database, one-third of entries contain at least one protein of known 3D structure. Pfam also contains functional annotation, literature references and database links for each family.

Hmmpfam, one program in the HMMER 2.2g package, is a tool for searching a single sequence against an HMM database. In real situations, this program may take a few weeks to a few months to process large amounts of sequence data. Thus efficient parallelization of the Hmmpfam is essential to bioinformatics research.

HMMER 2.2g provides a parallel Hmmpfam program based on PVM (Parallel Virtual Machine) [26]. However, the PVM version does not have good scalability and cannot fully take advantage of the current advanced supercomputing clusters. So a highly scalable and robust cluster-based solution for Hmmpfam is necessary. We implemented a parallel Hmmpfam harnessing the power of a multithreaded architecture and program execution model – the EARTH (Efficient Architecture for Running THreads) model [3, 4], where parallelism can be efficiently exploited on top of a supercomputing cluster built with off-the-shelf microprocessors.

The major contributions of this research are as follows: (1) the first EARTH-based parallel implementation of a bioinformatics sequence classification application; (2) a largely scalable parallel Hmmpfam implementation targeted to advanced supercomputing clusters; (3) the implementation of a new efficient master-slave dynamic load balancer in the EARTH runtime system. This load balancer is targeted to parallel applications adopting a master-slave model and shows more robust performance than a static load balancer.

The remainder of this chapter is organized as follows. In section 2.2, we review the Hmmpfam program and the original parallel scheme implemented on PVM, the EARTH model is reviewed in 2.3. Our cluster-based multithreaded parallel implementation is described in section 2.4 and section 2.5. The performance results of our implementation are presented in section 2.6, and conclusions in section 2.7.

2.2 HMMPFAM Algorithm and PVM Implementation

Hmmpfam reads a sequence file and compares each sequence within it, one at a time, against all the family profiles in the HMM database, looking for significantly similar matches. Fig. 2.3 shows the basic program structure of Hmmpfam. Fig. 2.4 shows the task space decomposition of the parallel scheme in the current PVM implementation. In this scheme, the master-slave model is adopted, and within one stage, all slave nodes work on the computation for the same sequence. The master node dynamically assigns one profile from the database to a specific slave node, and the slave node is responsible for the alignment of the sequence to this HMM profile. Upon finishing its job, the slave node reports the results to the master, which responds by assigning a new job, i.e. a new single profile, to that slave node. When all the computation of this sequence against the whole profile database is completed, the master node sorts and ranks the results it collects, and outputs the top hits. Then the computation on the next sequence begins.

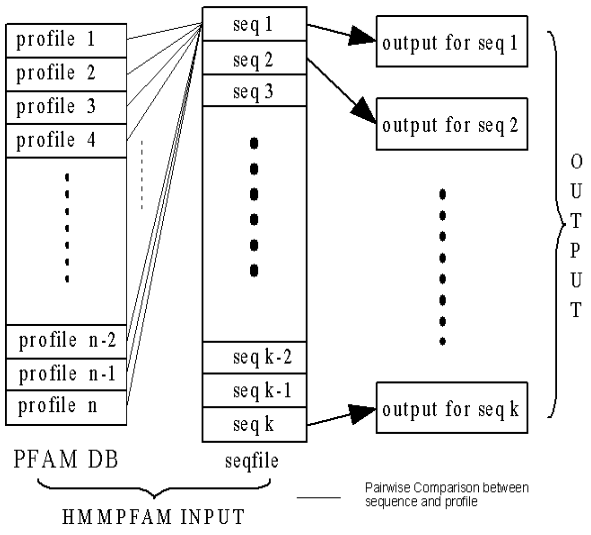


Figure 2.3: Hmmpfam program structure

The experimental results indicate that this implementation does not achieve good scalability as the number of computing nodes increases (Fig. 2.9). The problem is that the computation time is too small relative to the communication overhead. Moreover, the master node becomes a bottleneck when the number of the computing nodes increases, since it involves both communications with slave nodes and computations such as sorting and ranking. The implicit barrier at the end of the computation of one sequence also wastes the computing resources of the slave nodes.

2.3 EARTH Execution Model

The new parallel implementation of the Hmmpfam algorithm is based on EARTH multithreaded architecture, which is developed by the Computer Architecture and Parallel Systems Laboratory (CAPSL) at the University of Delaware. In this section, before presenting our implementations, we briefly describe EARTH, a parallel multithreaded architecture and execution model.

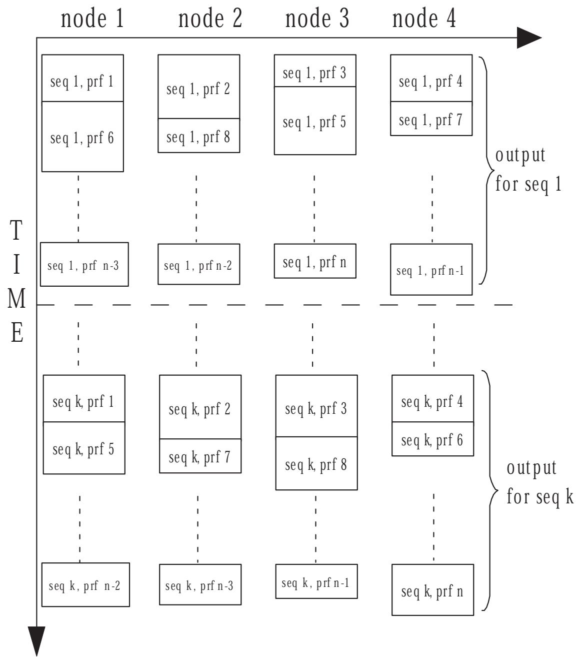


Figure 2.4: Parallel scheme of PVM version

EARTH (Efficient Architecture for Running THreads) [3, 4] supports a multi-threaded program execution model in which a program is viewed as a collection of threads whose execution ordering is determined by data and control dependencies explicitly identified in the program. Threads, in turn, are further divided into fibers which are non-preemptive and scheduled according to data-flow like firing rules, i.e., all needed data must be available before it becomes ready for execution. Programs structured using this two-level hierarchy can take advantage of both local synchronization and communication between fibers within the same thread, exploiting data locality. In addition, an effective overlapping of communication and computation is made possible by providing a pool of ready-to-run fibers from which the processor can fetch new work as soon as the current fiber ends and the necessary communication is initiated.

As shown in Fig. 2.5, an EARTH node is composed of an execution unit (EU), which runs the fiber, and a synchronization unit (SU), which schedules the fibers when

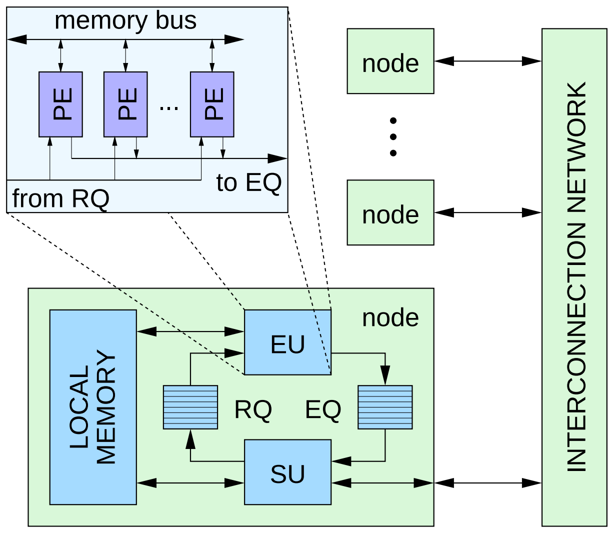


Figure 2.5: EARTH architecture

they are ready and handles the communication between nodes. There is also a ready queue (RQ) of ready fibers and an event queue (EQ) of EARTH operations generated by fibers running on EU. The EARTH architecture executes applications coded in Threaded-C [61], a multithreaded extension of ANSI-C programming language, which by incorporating EARTH operations, allows the programmer to indicate the parallelism explicitly. Although designed to deal with multiple threads per node, the EARTH model does not require any support for rapid context switching (since fiber is non-preemptive) and is well-suited to running on off-the-shelf processors. The EARTH Runtime System 2.5 (RTS 2.5) is implemented to support the execution of EARTH applications on Beowulf clusters that contain SMP nodes.

The EARTH RTS 2.5 [62–64] provides an interface between an explicitly multithreaded program and a distributed memory hardware platform [65]. It's portable on various platforms: x86-based Beowulf clusters, Sun SMP clusters, IBM SP2, etc. It performs fiber scheduling, inter-node communication, inter-fiber synchronization, global memory management, and an important feature – dynamic load balancing.

2.4 New Parallel Scheme

2.4.1 Task Decomposition

To efficiently parallelize an application, it is important to determine a proper task decomposition scheme. In parallel computing, we normally decompose a problem into many small tasks that run in parallel. A smaller task size means that relatively small amounts of computational work are done between communication events, which, in turn, implies a low computation-to-communication ratio and high communication overhead. A smaller task size, however, facilitates load balancing. We often use “granularity” as a qualitative measure of the ratio of computation-to-communication. Finer granularity means a smaller task size. The proper granularity depends on the algorithm and the hardware environment.

In the original scheme, the alignment of one sequence with one profile is treated as a single task. In order to reduce communication overhead, our scheme considers the computation of one sequence against the whole database as a single task. Normally the number of sequences in a sequence data file is much larger than the number of computing nodes available in current Beowulf clusters. So the number of single tasks is still relatively large to keep all nodes busy. Usually, the sequences are of similar length; thus we can also achieve good load balancing even with a bigger task size. Moreover, because the computation of one single sequence is performed by one process on one fixed node, the sorting and ranking can be done locally on that particular node, thus freeing the master from the burden of such computation.

2.4.2 Mapping to the EARTH Model

The EARTH model allows dynamic and hierarchical generation of threaded procedures and fibers, thus allowing us to use a two-level parallel scheme. At level one, as shown in Fig. 2.6, we map each task to a threaded procedure in the EARTH model. The threaded procedure is a C function containing local states (function parameters, local variables, and synchronization slots) and one or more fibers of the code. Either the

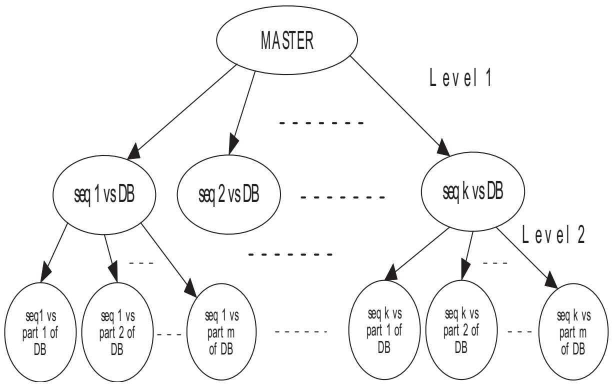


Figure 2.6: Two level parallel scheme

programmer or EARTH RTS can determine where (on which node) a procedure gets executed. At this level, the master process assigns each sequence to one and only one threaded procedure. Each procedure conducts the computation for the sequence against the whole HMM database, then sorts and ranks the alignment results, and outputs the top hits to a file on the local disk. The task size at this level is large and independent to other tasks at the same level, so this level exploits the coarse-grain parallelism.

The tasks of level one can be further divided into the smaller tasks of level two, each one of them conducting the comparison/aligment of one sequence versus a partition of the HMM database. Each task of level two can be mapped to a "fiber" in the EARTH model. Each fiber gets one partition of the database, performs computation, then returns the result to its parent procedure. This level exploits the fine-grain parallelism.

2.4.3 Performance Analysis

In this subsection, a comparison of the proposed new approach with the PVM approach is presented. The parameters and assumptions are listed as follows:

1. We have n profiles in the profile database and k sequences in the sequence file.

2. The computation of one sequence versus one profile takes the same amount of time, which is denoted as T_0.
3. Denote the time for one back and forth communication as T_c.
4. Assume that the master node can always respond to requests from slaves concurrently and immediately, and that the bandwidth is always sufficient; thus slaves have no idle waiting time.

In the original PVM approach, the basic task unit is computation of one sequence versus one profile. There is a total of $k \times n$ such tasks. Each one of them needs T_0 computation time and T_c communication time. Thus, the total work load (the sum of computation and communication) is:

$$WL = k \times n \times (T_0 + T_c) \tag{2.2}$$

In our new approach, one basic task unit is computation of one sequence versus the whole database, including n profiles. There is a total of k such tasks. Each task needs $n \times T_0$ computation time and T_c Communication time because only one communication is necessary for one task. Thus, the total work load is

$$WL = k \times (n \times T_0 + T_c) \tag{2.3}$$

The workload saved by our approach is:

$$WL_{save} = k \times (n - 1) \times T_c \tag{2.4}$$

From (2.4), it can be seen that a larger k and n indicate a larger improvement of our approach.

In addition to the reasons analyzed in the preceding formulas, there are several other factors that contribute to the better performance of our approach. Firstly, the master node in our approach has less chance of becoming a bottleneck. When the number of slave nodes is very large, a lot of requests from the slaves to the master may happen at the same time. Since the master node has to handle the requests one by one and the communication bandwidth of the master node is limited, the assumption of "immediate responses from the master" may not be valid anymore. As mentioned in Section 2.2, the PVM approach regards the computation of one sequence against one profile as a task, and the computation time for this task is very short, so the slave nodes send requests to the master very frequently. Our approach regards one sequence against the whole database as one task unit and has a larger computation time for each task unit; therefore the requests occurs less frequently. Thus, the chance of many requests blocked at the master node for the PVM approach is much higher than our approach. Secondly, since the computations of ranking and sorting are performed at the master node for the PVM approach, during this stage, all the slaves are idle. In our approach, however, the ranking and sorting are distributed to the slaves; thus the slaves have less idle time waiting for the response from the master node.

2.5 Load Balancing

We implemented the parallel scheme in Fig. 2.6 using two different approaches: the static and the dynamic load balancing. The static load balancing approach predetermines job distribution using the round-robin algorithm. The dynamic load balancing approach, in contrast, distributes tasks during execution with the load balancing support of the EARTH Runtime system.

2.5.1 Static Load Balancing Approach

In the static load balancing implementation shown in Fig.2.7, we explicitly spread out the tasks across the computing nodes before the execution of any process. To achieve an even work load, we adopted the round robin algorithm. During the initiation stage, the master node reads sequences one by one from the sequence file and generates new jobs for each of them by invoking a threaded procedure on the specified node. The EARTH

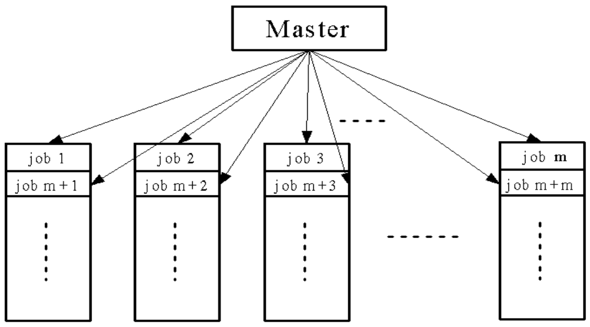


Figure 2.7: Static load balancing scheme

RTS then puts all the invoked threaded procedures into a ready queue for each slave node. During the computation stage, each slave node fetches jobs from its own ready queue, which means all nodes execute jobs without frequent communication with the master node. A sequence file contains a large amount of sequences which are usually of similar length, so the static approach can achieve an evenly balanced work load and good scalability.

2.5.2 Dynamic Load Balancing Approach

The EARTH RTS includes an inherent dynamic load balancing mechanism, which collects information on the dynamic system status to conduct run-time workload dispatching. The design of the dynamic load balancer focuses on two objectives: (1) keeping all the nodes busy; (2) minimizing the overheads of load balancing.

In fact, the research on the parallelization of Hmmpfam motivated us to design a load balancer in the EARTH RTS 2.5, as illustrated in Fig. 2.8. With the dynamic load balancing support of the EARTH RTS, the job distribution is completely transparent to programmers. The EARTH RTS takes over the responsibility of dispatching jobs at the runtime, which makes programming much simpler. The RTS maintains a ready queue at

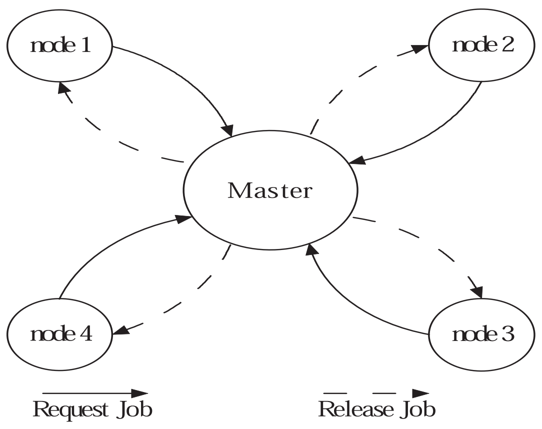


Figure 2.8: Dynamic load balancing scheme

the master node and sends tasks to slave nodes one by one during the execution. Once a slave node finishes a job, it requests another task from the EARTH RTS on the master node.

The dynamic load balancing approach is more robust than the pre-determined job assignments strategy. In the static load balancing approach, all jobs are put into the ready queue of slave nodes during the initiation stage, and cannot be moved away after that. If one node has a heavier work load than others or even stops working, its jobs cannot be reassigned to other nodes. The dynamic load balancing strategy, in contrast, is able to avoid this situation because the EARTH RTS maintains the ready queue at the master node. The robustness of Hmmpfam makes an important issue considering the fact that Hmmpfam may run for quite a long time (e.g., several weeks). Also, on a supercomputing cluster that consists of hundreds of computing nodes, a robust approach becomes necessary because it is not easy to guarantee that all nodes work properly without any failure.

2.6 Experimental Results

2.6.1 Computational Platforms

The experiments described in this research are carried out by using the EARTH Runtime System 2.5 and three different Beowulf clusters. The comparison of the PVM Hmmpfam version and the EARTH version is tested on the COMET cluster at the Computer Architecture and Parallel Systems Laboratory (CAPSL) of the University of Delaware. COMET consists of 18 nodes: each node has two 1.4 GHz AMD Athlon processors and 512MB of DDR SDRAM memory. The interconnection network for the nodes is a switched 100Mbps ethernet.

Other experiments are conducted on two large clusters. The Chiba City cluster [66] is a scalability testbed at the Argonne National Laboratory. The cluster is comprised of 256 computational servers, each with two 500MHz Pentium III processors and 512MB RAM memory. The interconnects for high performance communication include a fast ethernet and a 64-bit Myrinet.

The JAZZ [67] cluster is a teraflop-class computing cluster at the Argonne National Laboratory. It consists of 350 computing nodes, each with a 2.4 GHz Pentium Xeon processor. All nodes are interconnected by fast ethernet and Myrinet 2000. Detailed configuration of the platforms is summarized in Table 2.1.

Table 2.1: Experiment platforms

Name	Location	Processor type	# of CPUs	Memory per node	Network
Comet	UDel	AMD Athlon 1.4G	18×2 per node	512M	100T Ethernet
Chiba City [66]	ANL	PIII 500MHz	256×2 per node	512M	Gigabit Ethernet
JAZZ [67]	ANL	Xeon 2.4GHz	350×1 per node	2G/1G	Gigabit Ethernet

2.6.2 Experimental Benchmarks

For the comparison of the PVM version and the EARTH version of parallel Hmmpfam, we use an HMM database containing 585 profile families, and a sequence file with 250 sequences. This benchmark is referred to as data set-1 in the following sections. Data set-1 is also used in the robustness experiment.

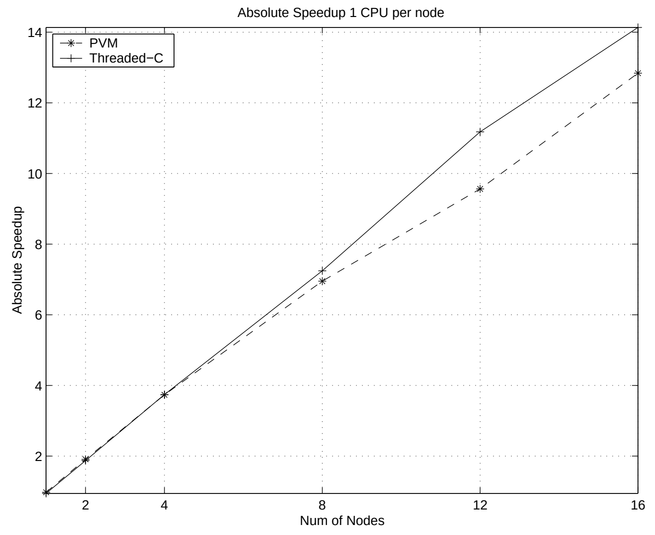
For testing both the static and dynamic load balancing version of EARTH Hmmpfam, we use an HMM database containing 50 profile families, and a sequence file containing 38192 sequences. This benchmark is referred to as data set-2 in the following sections.

2.6.3 Comparison of PVM-based and EARTH-based Implementations

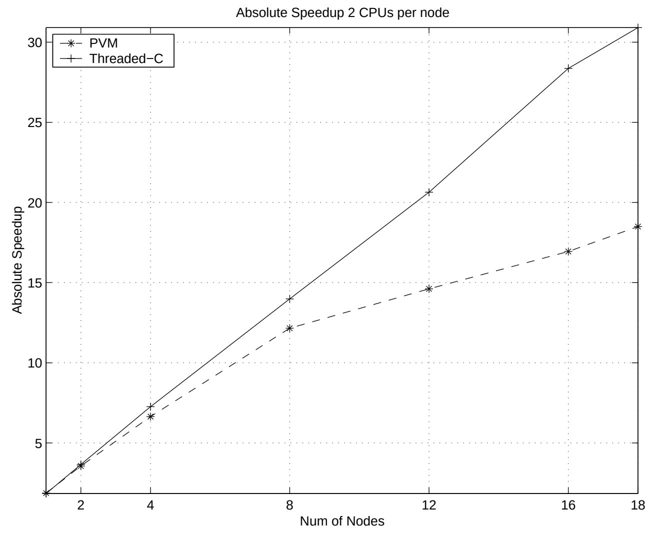
The first test is conducted to compare the scalability of the PVM version and the EARTH version on the COMET cluster using test data set-1. Fig. 2.9a shows the absolute speedup curve when both the PVM version and the EARTH version are configured to use only 1 CPU per node in COMET, while Fig. 2.9b shows the results for dual CPUs per node configuration. From the figures, it is easily seen that the proposed new version has much better scalability, especially in dual-CPU per node configuration. For example, with 16 nodes and 2 CPUs per node configuration, the absolute speedup of the PVM version is 18.50, while the speedup of our version is 30.91, which means 40% reduction of execution time. This is due to the fact that our implementation increases the computation granularity and avoids most communication costs and internal barriers.

2.6.4 Scalability on Supercomputing Clusters

The second and third tests are conducted to show the performance of our EARTH version Hmmpfam on large clusters, the Chiba City cluster and the JAZZ cluster, using test data set-2. The results of both static load balancing and dynamic load balancing schemes are shown in Fig. 2.10 to Fig. 2.11, where Fig. 2.10a and Fig. 2.10b show the results for static load balancing on 1 CPU per node and 2 CPUs per node configuration, and Fig. 2.11a and Fig. 2.11b are the results for dynamic load balancing. The two methods do not have much difference in the absolute speedup. This is due to the fact that subtasks are relatively similar in size, which means static load balancing can also

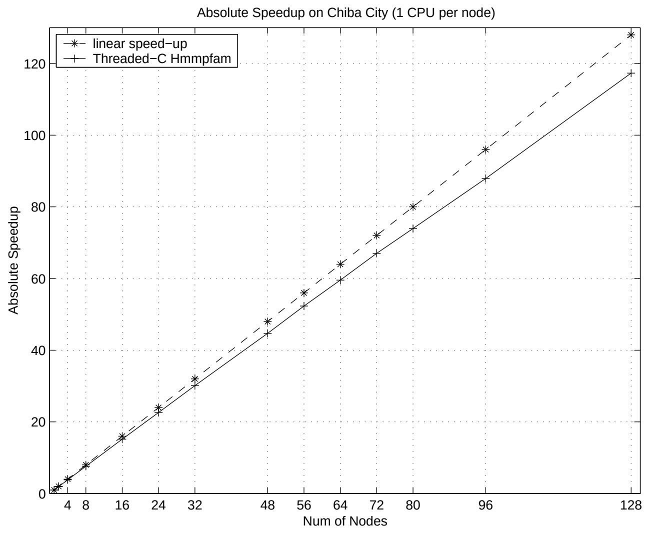


(a)

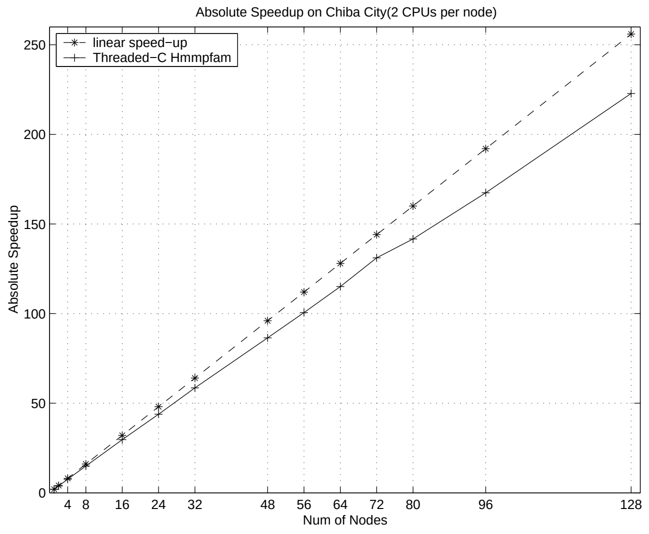


(b)

Figure 2.9: Comparison of PVM and EARTH based implementations (a) 1 CPU per node (b) 2 CPUs per node

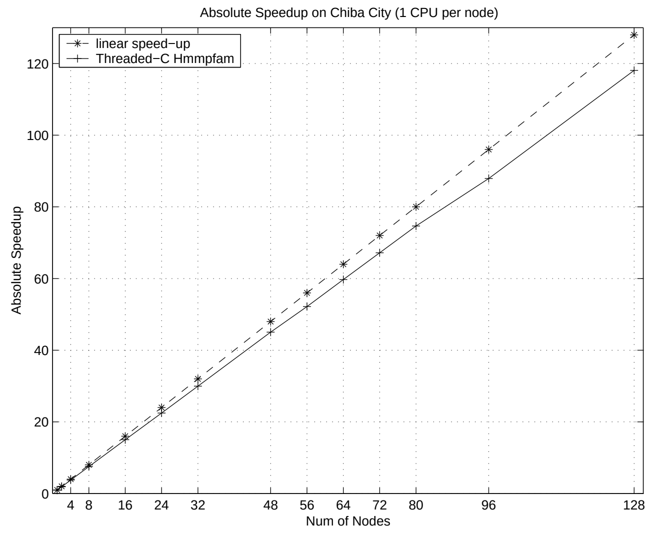


(a)

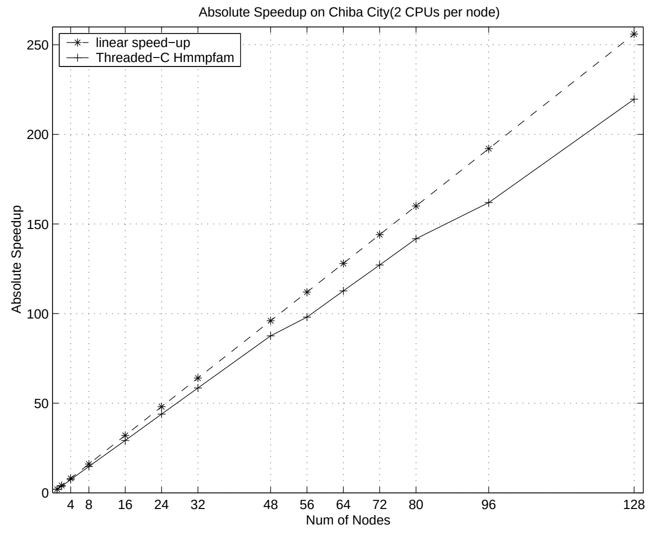


(b)

Figure 2.10: Static load balancing on Chiba City (a) 1 CPU each node (b) 2 CPUs each node



(a)



(b)

Figure 2.11: Dynamic load balancing on Chiba City (a) 1 CPU each node (b) 2 CPUs each node

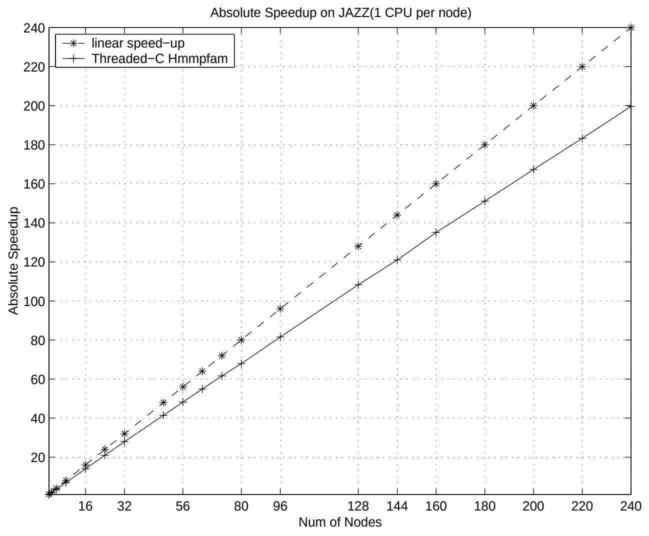


Figure 2.12: Dynamic load balancing on JAZZ

achieve good performance. Both of them show a near linear speedup, which means in our new parallel scheme, the serial part only occupies a very small percentage of the total execution. As long as the test data set is big enough, the speedup is expected to keep near linear up to 128 nodes on the the Chiba City Cluster. The test results on the JAZZ cluster are shown in Fig. 2.12. The speedup curve shows that our implementation can get a near linear speedup on 240 nodes.

2.6.5 Robustness of Dynamical Load Balancing

One of the advantages of the dynamic load balancing approach is its robustness. The experiments are conducted to show that the program with dynamic load balancing is less affected by the disturbance (the resource contention caused by other applications running at the same time). The Blastall [42] program is used as the disturbance source since this program is another commonly used computation-intensive bioinformatics software.

The execution time for both the static and the dynamic approaches with and without disturbance is measured. Let T denote the execution time without disturbance, and

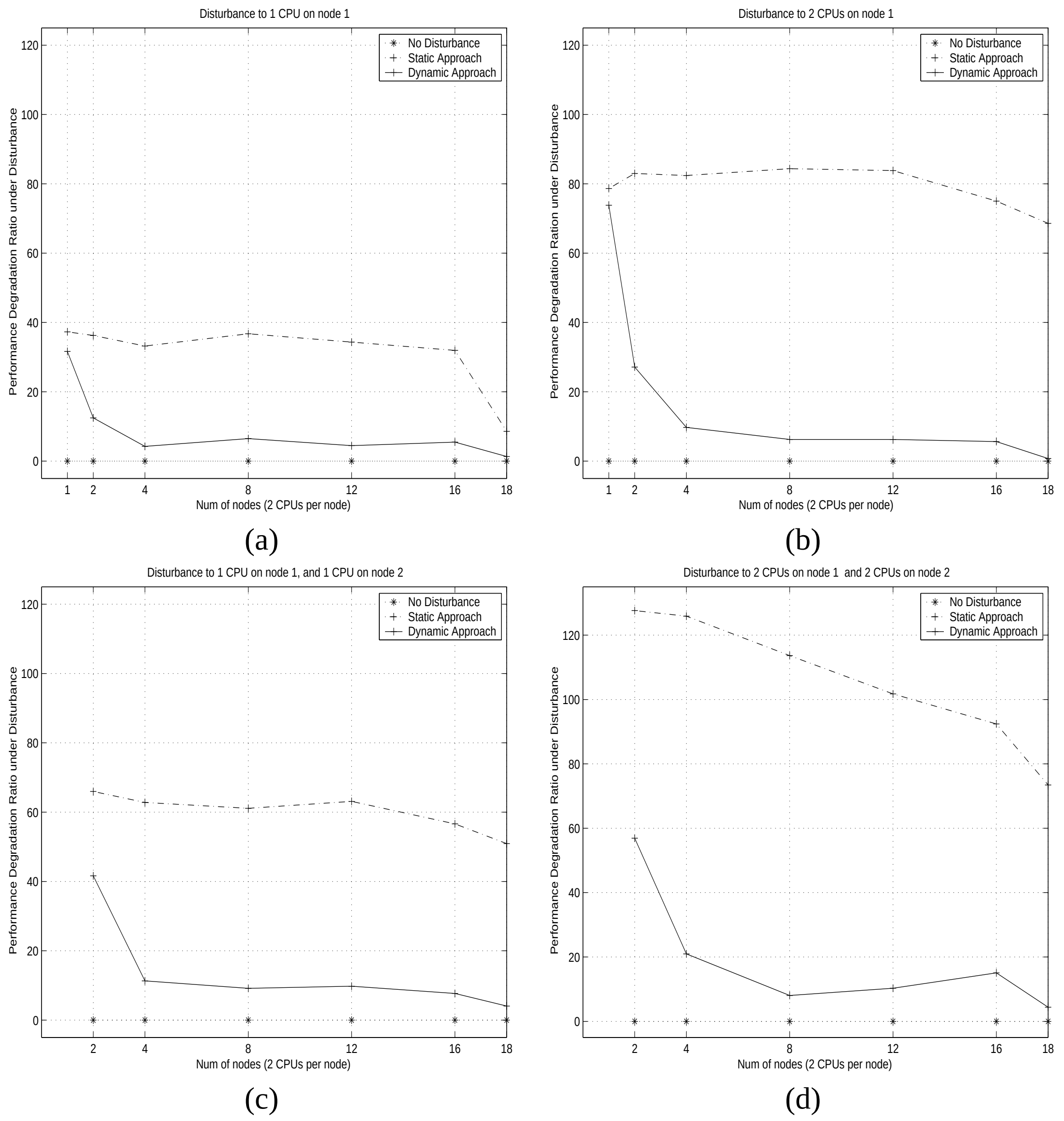


Figure 2.13: Performance degradation ratio under disturbance (a) disturbance to 1 CPU (b) disturbance to 2 CPUs on 1 node (c) disturbance to 2 CPUs on 2 nodes (d) disturbance to 4 CPUs on 2 nodes

T' denote the execution time with disturbance. Define the *performance degradation ratio under disturbance* (PDRD) as:

$$PDRD = (\frac{T' - T}{T}) \times 100\% \tag{2.5}$$

The PDRD is computed and plotted for both the static and the dynamic approaches. A smaller PDRD indicates that the performance is less influenced by the introduction of disturbance, thus implying the higher implementation robustness.

For the robustness experiment, the data set-1 is used on the COMET cluster. Fig. 2.13a shows the result when only one Blastall program is running on 1 CPU to disturb the execution of Hmmpfam, and Fig. 2.13b shows the result when two CPUs of one node are both disturbed. Fig. 2.13c and Fig. 2.13d show the result when 2 computing nodes are disturbed. From the figures, it is apparent that the dynamic load balancing program is less affected by the disturbance and thus has higher robustness.

2.7 Summary

We implemented a new cluster-based solution of the HMM database searching tool on the EARTH model and demonstrated significant performance improvement over the original parallel version based on PVM. Our solution provides near linear scalability on supercomputing clusters. Comparison between the static and dynamic load balancing approaches shows that the latter is a more robust and practical solution for large-scale time-consuming applications running on clusters.

This new implementation allows researchers to analyze biological sequences at a much higher speed and also makes it possible for scientists to analyze problems that were previously considered too large and too time consuming. The parallelization implementation in this work motivated the addition of a robust dynamic load balancing support into the EARTH model, which proves that applications could be the driving force for design of architecture and programming models.

Chapter 3

SPACE RIP TARGETED TO CELLULAR COMPUTER ARCHITECTURE CYCLOPS-64

3.1 Introduction

This chapter presents the parallelization and performance optimization of another biomedical application–SPACE RIP, a parallel imaging technique on the Cyclops-64 multiprocessor-on-a-chip computer architecture. Cyclops-64 [10–12, 34, 35] is a new architecture being developed at the IBM T. J. Watson Research Center and the University of Delaware. SPACE RIP (Sensitivity Profiles From an Array of Coils for Encoding and Reconstruction in Parallel) is one of the parallel imaging techniques which use spatial information contained in the component coils of an array to partially replace spatial encoding which would normally be performed using gradients in order to reduce imaging acquisition time. We present the parallelization and optimization of SPACE RIP at three levels. The top level is the loop level parallelization. The loop level parallelization decomposes SPACE RIP into many SVD problems. This is possible because the reconstructions of each column in an image are independent of each other. The reconstruction of each column is a pseudoinverse of a matrix, which is solved by the singular value decomposition (SVD). The middle level is the parallelization of a SVD problem using one-sided Jacobi algorithm and is implemented on Cyclops-64. At this level, an SVD problem is decomposed into many tasks, each one of them is a matrix column rotation routine. The bottom level further optimizes the matrix column rotation routine by using several memory preloading or loop unrolling approaches.

1. We implemented the parallelization and optimization of SPACE RIP at three levels. The top level is the loop level parallelization, which decomposes SPACE RIP into many tasks of a singular value decomposition (SVD) problem. The middle level parallelizes the SVD problem using the one-sided Jacobi algorithm and is implemented on Cyclops-64. At this level, an SVD problem is decomposed into many matrix column rotation routines. The bottom level further optimizes the matrix column rotation routine using several memory preloading or loop unrolling approaches.

2. We developed a model and trace analyzer to decompose the total execution cycles into four parts: total instruction counts, “DLL”, “DLF” and “DLI”, where “DLL” represents the cycles spent on memory access, “DLF” represents the latency cycles related to floating point operations, and “DLI” represents the latency cycles related to integer operations. This simple model allows us to study the application performance tradeoff for different algorithms.

3. Using a few application parameters such as matrix size, group size, and architectural parameters such as onchip and offchip latency, we developed analytical equations for comparing different memory access approaches such as preloading and loop unrolling. We used a cycle accurate simulator to validate the analysis and compare the effect of different approaches on the “DLL” part and the total execution cycles.

The remainder of this chapter is organized as follows. The target platform Cyclops-64 is introduced in Section 3.2. The background of MRI imaging is presented in Section.3.3. The SPACE RIP technique is briefly reviewed in section 3.4 to expose the parallelism inherent in the problem. The coarse grain loop level parallelization is presented in Section 3.5, and the fine grain parallelization of the SVD algorithm is presented in Section 3.6. Different memory access approaches are introduced in Section 3.7 in order to further improve the performance of the rotation routine of the SVD algorithm. Detailed analysis of these approaches is presented in Section 3.8. The performance experimental results are shown in Section 3.9 and the conclusions summarized in Section 3.10.

3.2 Cyclops-64 Hardware and Software System

The Cyclops-64 project is a petaflop supercomputer project. The main principles of the Cyclops-64 architecture [10] are: (1), the integration of processing logic and memory in a single piece of silicon; (2), the use of massive intra-chip parallelism to tolerate latencies; (3) a cellular approach to building large systems; (4), the smaller inter-processor communication and synchronization overhead brings better performance.

The Cyclops-64 system is a general purpose platform that can support a wide range of applications. Some possible kernel applications include FFT and other linear algebra such as BLAS 1 and 2 of LAPACK [68] package, protein folding and other bioinformatics applications. In this research, Cyclops-64 is adopted for solving the SVD linear algebra problem in the context of biomedical imaging.

Fig. 3.1 shows the hardware architecture of a Cyclops-64 chip (a.k.a C64). One Cyclops-64 chip has 80 processors, each consisting of two thread units, a floating-point unit, and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. In a Cyclops-64 chip architecture there is no data cache. Instead, a portion of each SRAM bank can be configured as scratch-pad memory. Such a memory provides a fast temporary storage to exploit locality under software control.

On the software side, one important part of the Cyclops-64 system software is the Cyclops-64 thread virtual machine. It is worth noting that an OS is not developed. Instead, CThread (Cyclops-64 thread) is implemented directly on top of the hardware architecture as a micro-kernel/run-time system that fully takes advantage of the Cyclops-64 hardware features.

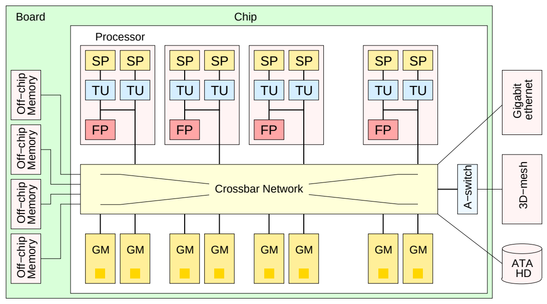


Figure 3.1: Cyclops-64 chip

The Cyclops-64 thread virtual machine includes a thread model, a memory model and a synchronization model. The Cyclops-64 chip hardware supports a shared address space model: all onchip SRAM and offchip DRAM banks are addressable from all thread units/processors on the same chip, which means that all the threads can see a single shared address space. More details are explained in [34, 35, 69].

In the thread synchronization model, the CThread mutex lock and unlock operations are directly implemented using Cyclops-64 hardware atomic test-and-set operations and are thus very efficient. Furthermore, a very efficient barrier synchronization primitive is provided. Barriers are implemented using the "Signal Bus" special purpose register. The barrier function can be invoked by a group of threads. Threads will block until all the threads participating in the operation have reached this routine.

The memory organization is summarized in Table 3.1. The default offchip latency is 36 cycles. It can become larger when there is a heavy load of memory accesses from many thread units. This parameter can be preset in the Cyclops-64 simulator. In this experiment, the offchip latency is set to be 36 or 80.

Another set of parameters in the Cyclops-64 simulator is the delay of instructions. The delay for an instruction is decomposed in two parts, execution cycles and latency cycles. The execution unit is kept busy for the number of execution cycles and another instruction cannot be issued during the execution cycles. The result is available after the number of execution+latency cycles. The resources can, however, be utilized by other

Table 3.1: Memory configuration

Memory position	size (Byte)	Latency (cycle)
scratch-pad	$80 \times 2bank \times 16K$	2
onchip SRAM	$80 \times 2bank \times 16K$	19
offchip DRAM	$4bank \times 512M$	36

instructions during the latency period.

3.3 MRI Imaging Principles

In this section, the MRI (Magnetic resonance imaging) physics and basic concepts such as "frequency encoding", "phase encoding" and "k" space are reviewed. MRI is a method of creating images of the inside of opaque organs in living organisms. It is primarily used to demonstrate pathological or other physiological alterations of living tissues and is a commonly used form of medical imaging.

Paul Lauterbur and Sir Peter Mansfield were awarded the 2003 Nobel Prize in Medicine for their discoveries concerning MRI. Lauterbur discovered that gradients in the magnetic field could be used to generate two-dimensional images. Mansfield analyzed the gradients mathematically. The Nobel Committee ignored Raymond V. Damadian, who demonstrated in 1971 that MRI can detect cancer and filed a patent for the first whole-body scanner.

3.3.1 Larmor Frequency

MRI is founded on the principle of nuclear magnetic resonance (NMR), which is shown in Fig. 3.2. There is electric charge on the surface of the proton, thus creating a small current loop and generating magnetic moment M. The proton also has mass which generates an angular momentum when it is spinning. If the proton is put into a magnetic field B_0, the magnetic field causes M to rotate (or precess) about the direction of B_0 at

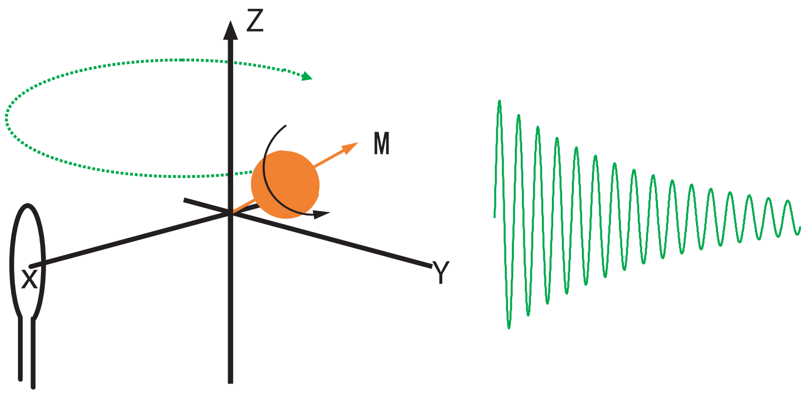


Figure 3.2: Proton rotation and the induced signal

a frequency proportional to the magnitude of B_0, which is called Larmor frequency [70]. Conventionally, the Larmor equation is written as:

$$\omega_0 = \gamma B_0, \tag{3.1}$$

where ω_0 is the angular frequency of the protons ($\omega = 2\pi f$). Using this scheme gives γ a value of 2.67×10^8 radians $s^{-1}T^{-1}$. When the use of scale frequency is helpful, we use $\bar{\gamma}$ (gamma bar), which is equal to $\gamma/2\pi$ (i.e. $42MH_zT^{-1}$). Thus the scalar frequency is given by:

$$f_0 = 42 \times B_0. \tag{3.2}$$

As the transverse component (the component in the x, y plane) of M rotates about the z axis, it will induce a current in a coil of wire located around the x axis, as shown in Fig. 3.2. This signal collected by the coil is the free induction signal (FID). The frequency of the induced signal is the Larmor frequency. The induced signal is used for the MRI imaging.

The measured MR signal is the net signal from the entire object, which is calculated by integrating transverse magnetization along x:

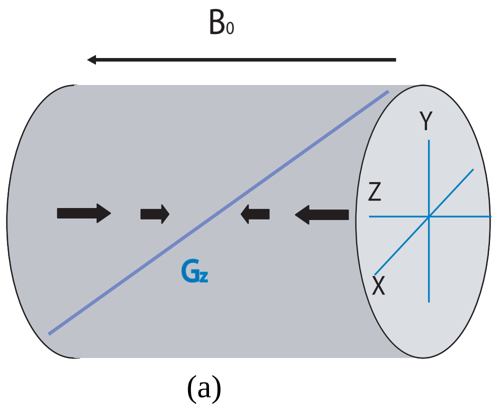
$$S = \int_{-\infty}^{\infty} M(x)dx, \tag{3.3}$$

where $M(x) = r(x)e^{j\theta(x)}$. $r(x)$ is the density of magnetization along x, and $\theta(x)$ is the local phase angle at x. This signal alone is not able to produce an image since there is no way to tell where the signal comes from [71]. Thus the frequency and phase encoding gradient is necessary for encoding position information.

3.3.2 Frequency Encoding and Phase Encoding

MRI use frequency and phase encoding to generate a 2D image. Both of them use magnetic field “gradient”, which refers to an additional spatially linear variation in the static field strength. Without gradient, the main magnetic field B_0 is homogenous. An “x gradient” will add to or subtract from the magnitude of the static field at different points along the x axis. Similarly, a “y gradient” and “z gradient” will cause a variation of magnitude along the y axis and z axis, respectively. The “z gradient” and “y gradient” are shown in Fig. 3.3 (Adapted from [70]). The “x gradient” is not shown due to the similarity between the “x” and “y” gradient, the only difference being the axis along which the magnetic field varies. The length of the vectors represents the magnitude of the magnetic field, which sometimes can also be represented by the density of the magnetic field line. The symbols for a magnetic field gradient in the x, y, and z directions are G_x, G_y, and G_z. Note that the “gradient” only changes the magnitude and does not change the direction, which is always along the z axis (B_0 direction). Conventionally, the z gradient is used for slice selection. The x gradient and y gradient are used for frequency encoding and phase encoding.

G_x has no effect on the center of the field of view ($x = 0$) but causes the total field to vary linearly with x, causing the resonance frequency to be proportional to the x position of the spin, as shown in Fig. 3.4 (Adapted from [70]). The slope of the straight line in Fig. 3.4 is equal to G_x. This procedure is called “frequency encoding” since the x position is encoded into the precession frequency. After precessing for a time t in this gradient field, the magnetization of a spin at position x will acquire an additional phase



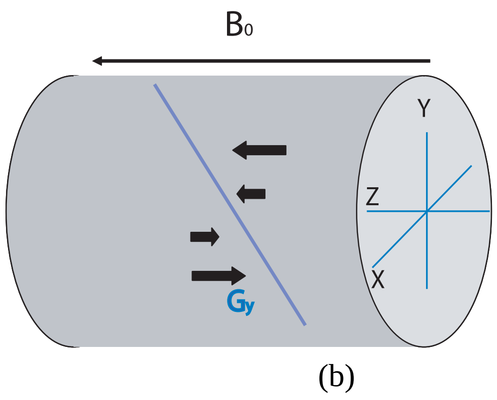


Figure 3.3: Encoding gradient (a) G_z and (b) G_y

$\theta_x = \gamma x G_x t$, and the measured signal at time t becomes:

$$S(t) = \int_{-\infty}^{\infty} r(x) e^{j\gamma G_x t} dx. \tag{3.4}$$

Equation 3.4 is the form of an inverse Fourier transform. The frequency encoding gradient is applied continuously "during" the signal acquisition and generates 1D imaging.

In order to get a second dimension, an additional gradient "G_y" is introduced. It is applied with a duration of τ "prior" to the signal measurement. Thus the magnetization of a spin at position y will get an additional phase $\theta_y = \gamma y G_x \tau$. This process is called "phase encoding" since the y position is encoded as an additional phase before measurement. With both frequency encoding and phase encoding, the measured signal becomes:

$$S(t) = \int\int r(x,y) e^{j\gamma x G_x t} e^{j\gamma y G_y \tau} dx dy. \tag{3.5}$$

This Equation is in the form of 2D inverse Fourier transform and forms the base for 2D imaging.

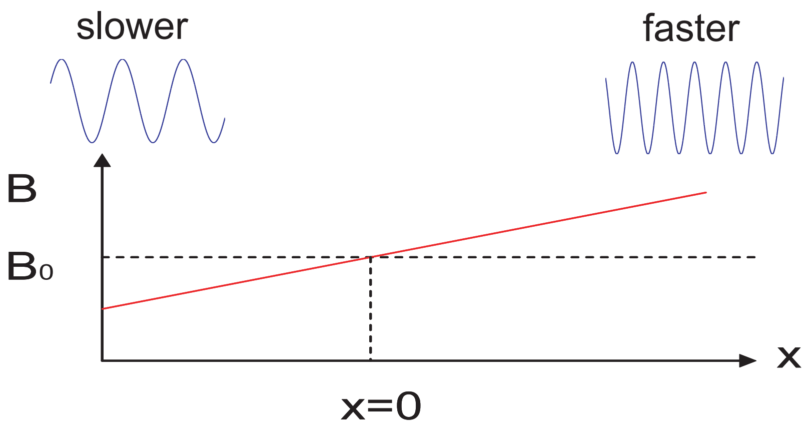


Figure 3.4: Effect of field gradient on the resonance frequency

3.3.3 k Space and Image Space

In a complete MR acquisition, the signals are sampled M times at intervals Δt, and the phase encoding gradient pulse sequence repeated N times, each time incrementing the phase encoding gradient amplitude such that

$$G_{PE}z(n) = \Delta G \times n, \qquad for \quad n = -\frac{N}{2} to \frac{N}{2} - 1. \tag{3.6}$$

During each repetition, data are acquired and put into one horizontal line of the grid shown in Fig. 3.5 (adapted from [72]). In this figure, the “frequency encoding” and “phase encoding” directions are illustrated. Each time we change the phase encoding gradient, we acquire another line of data. The low phase encoding lines are written in the center of the grid, while the high phase encoding lines are written to the edges of the grids. Conventionally, we refer to the acquired data in the grid as “raw” data.

We define quantities k_{FE} and k_{PE} such that

$$k_{FE} = \bar{\gamma} \times G_x \times \Delta t \times m \tag{3.7}$$

$$k_{PE} = \bar{\gamma} \times \Delta G \times n \times \tau. \tag{3.8}$$

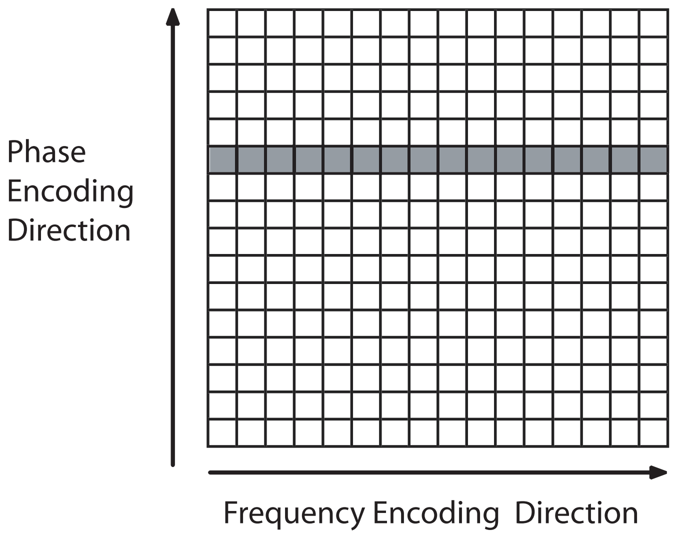


Figure 3.5: Frequency and Phase Encoding Direction

Then the total signal acquired in two dimensions time t and and "pseudo-time" τ is:

$$S(m,n) = \int\int r(x,y)e^{j2\pi xk_{FE}}e^{j2\pi yk_{PE}}dxdy, \tag{3.9}$$

which is in the form of an inverse Fourier transform of the spin density $r(x,y)$. The 2D FT of the encoded signal results (k-space raw data) in a representation of the spin density distribution in two dimensions (image space or coordinate space). The relation of the k-space and the image space is shown in Fig. 3.6. An example of an MRI image and its k-space amplitude are shown in Fig. 3.7. The central portion of k-space corresponds to the low spatial frequency components, and the outer edges describe the high frequencies.

3.4 Parallel Imaging and SPACE RIP

In the conventional serial imaging sequences, only one receiver coil is used to collect all the data; the phase encoding gradient G_y is varied in order to cover all of the k-space line with the desired resolution. One echo is needed for each value of G_y^g, making sequential imaging a time consuming procedure.

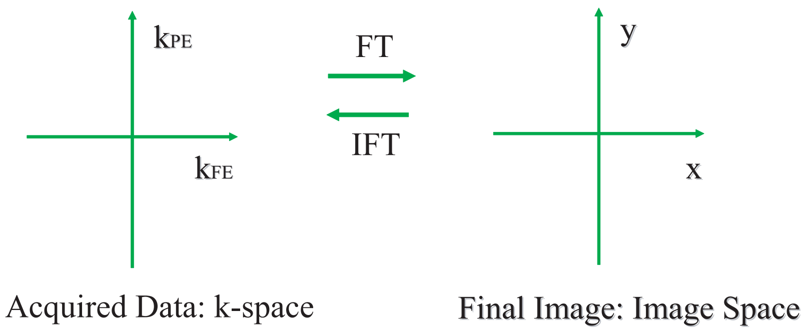


Figure 3.6: Relationship of k-space and image space

Reduction in acquisition time can reduce or even avoid motion artifacts, make the MR imaging more efficient and make it useful for more potential applications. For instance, dynamic imaging applications of cardiac contraction require high temporal resolutions without undue sacrifices in spatial resolution [73]. There are many ways to reduce the acquisition time for sequential imaging. For instance, multi-echo imaging EPI (Echo Planar Imaging) can achieve higher speed by optimizing strengths, switching rates, and patterns of gradients and RF (Radio Frequency) pulses. However, these approaches will sometimes decrease SNR (Signal to Noise ratio) or spatial resolution; also, they tend to require higher magnetic field strengths and increased gradient performance, thus reaching the technical limits.

Parallel imaging is based on using multiple receiver coils, each providing independent information about the image. Fig. 3.8 (adapted from [74]) shows a configuration with two coils. The sensitivity profile of the two coils and the coil views are shown in the second column and the third column of the figure, respectively. The parallel imaging techniques use spatial information contained in the component coils of an array to partially replace spatial encoding which would normally be performed using gradients, thereby reducing imaging acquisition time.

The name "parallel" is due to the fact that multiple MR signal data points are

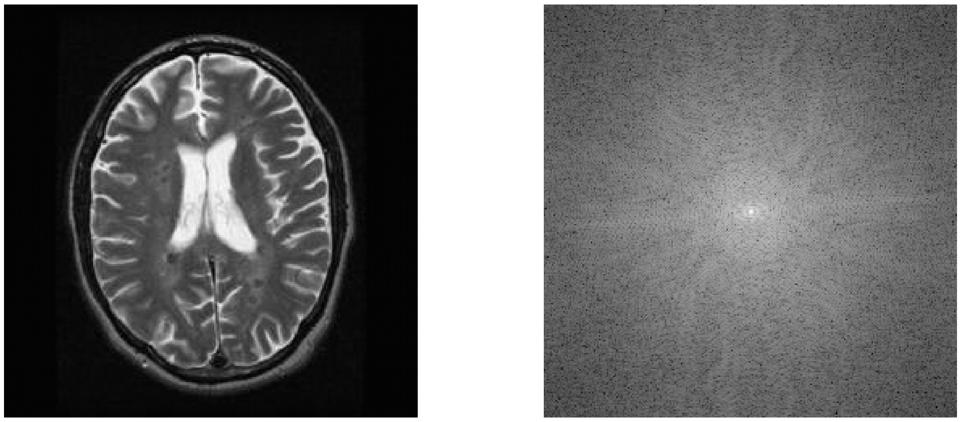


Figure 3.7: Example of image space and k space

acquired simultaneously. The maximum acquisition time reduction factor is the number of coils used. In a typical parallel imaging acquisition, only a fraction of the phase encoding lines are acquired compared to the conventional acquisition. Therefore, the k space is under-sampled, which causes the aliasing in the acquired coil views (aliased version of the second column of Fig. 3.8). A specialized reconstruction is applied to the acquired data to reconstruct the image.

There are three approaches of parallel imaging, known as SMASH [75], SENSE [73],and SPACE-RIP [76]. SMASH (SiMultaneous Acquisition of Spatial Harmonics) is a k-space domain implementation of the parallel imaging. It is based on the computation of the sensitivity profiles of the coils in one direction. These profiles are then weighted appropriately and combined linearly in order to form sinusoidal harmonics which are used to generate the k-space lines that are missing due to undersampling.

SENSE (sensitivity encoding) [73] is an image domain sensitivity encoding method. It relies on the use of 2D sensitivity profile information in order to reduce image acquisition time. Like SMASH, the cartesian version of SENSE requires the acquisition

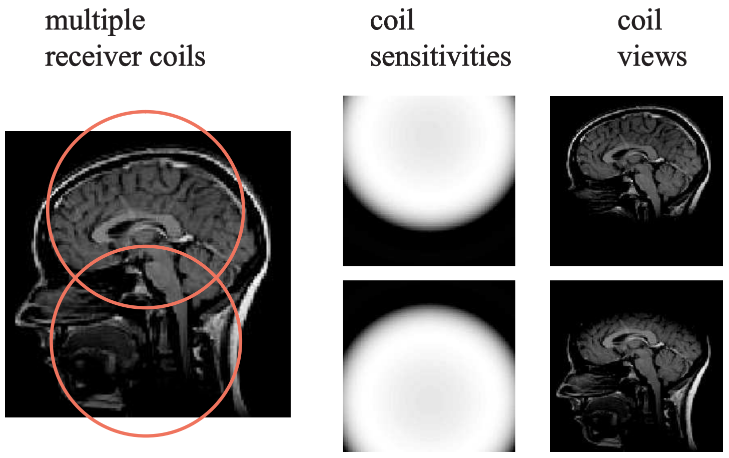


Figure 3.8: Example of coil configuration and coil sensitivity

of equally spaced k-space lines in order to reconstruct sensitivity weighted, aliased versions of the image. It is shown in [73] that the SENSE technique can reduce the scan time to one-half using a two-coil array in brain imaging and that double-oblique heart images can be obtained in one-third of conventional scan time with an array of five coils.

SPACE RIP [76] is the latest of the three methods. It uses k-space target data as input in conjunction with a real space representation of the coil sensitivities to directly compute a final image domain output. It generalizes the SMASH approach by allowing the arbitrary placement of RF receiver coils around the object to be imaged. It also allows any combination of k-space lines as opposed to regularly spaced ones. SPACE RIP has a higher computational burden than either SENSE or SMASH.

Fig. 3.9 shows the schematic representation of SPACE RIP acquisition and reconstruction. S1, S2, S3 and S4 are acquired data from four coils. The matrix G is the system gain matrix constructed from coil sensitivity profiles. I is the image to be constructed. The construction of the G matrix is explained as follows.

The MR signal received in a coil having $W_k(x, y)$ as its complex 2D sensitivity profile can be written as:

$$s_k(G_y^g, t) = \int\int r(x,y)W_k(x,y)e^{j\gamma(G_x x t + G_y^g y\tau)}dxdy, \tag{3.10}$$

where $r(x,y)$ denotes the proton density function, $W_k(x,y)$ is the complex 2D sensitivity profile of this coil, G_x represents the readout gradient amplitude applied in the x direction, G_y^g represents the phase encoding gradient applied during the g^{th} acquisition, x and y represent the x and y directions, respectively, τ is the pulse width of the phase encoding gradient G_y^g, and γ is a constant with the value of 2.67×10^8 radians $s^{-1}T^{-1}$.

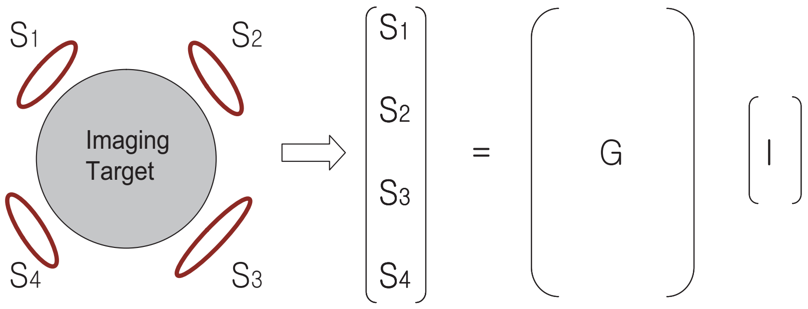


Figure 3.9: Schematic representation of SPACE RIP

Taking the Fourier transform of (3.10) along the x direction with a phase encoding gradient G_y^g applied yields:

$$S_k(G_y^g, x) = \int r(x,y)W_k(x,y)e^{j\gamma(G_y^g y\tau)}dy, \tag{3.11}$$

which is the phase modulated projection of the sensitivity weighted image onto the x axis. Here the x and y are continuous values. In order to obtain a discrete version of $r(x,y)$, $r(x,y)$ and $W_k(x,y)$ are expanded along the y direction utilizing a set of orthonormal sampling functions $\Psi_n(y)$. Further mathematical simplification [76] yields:

$$S_k(G_y^g, x) = \sum_{n=1}^{N} \eta(x,n)W_k(x,n)e^{j\gamma(G_y^g n\tau)}. \tag{3.12}$$

$$\begin{pmatrix} S_1(G_y^1,x) \\ \cdot \\ S_1(G_y^F,x) \\ \\ S_2(G_y^1,x) \\ \cdot \\ S_2(G_y^F,x) \\ \\ \cdot \\ \cdot \\ S_K(G_y^1,x) \\ \cdot \\ S_K(G_y^F,x) \end{pmatrix} = \begin{pmatrix} W_1(x,1)e^{j\gamma(G_y^1 1\tau)} & \cdots & W_1(x,N)e^{j\gamma(G_y^1 N\tau)} \\ \cdot & \cdots & \cdot \\ W_1(x,1)e^{j\gamma(G_y^F 1\tau)} & \cdots & W_1(x,N)e^{j\gamma(G_y^F N\tau)} \\ \\ W_2(x,1)e^{j\gamma(G_y^1 1\tau)} & \cdots & W_2(x,N)e^{j\gamma(G_y^1 N\tau)} \\ \cdot & \cdots & \cdot \\ W_2(x,1)e^{j\gamma(G_y^F 1\tau)} & \cdots & W_2(x,N)e^{j\gamma(G_y^F N\tau)} \\ \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ W_K(x,1)e^{j\gamma(G_y^1 1\tau)} & \cdots & W_K(x,N)e^{j\gamma(G_y^1 N\tau)} \\ \cdot & \cdots & \cdot \\ W_K(x,1)e^{j\gamma(G_y^F 1\tau)} & \cdots & W_K(x,N)e^{j\gamma(G_y^F N\tau)} \end{pmatrix} \cdot \begin{pmatrix} \eta(x,1) \\ \eta(x,2) \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \cdot \\ \eta(x,N) \end{pmatrix} \tag{3.13}$$

where N is the number of pixels in the y direction. $\eta(x,n)$ is the discretized version of $r(x,y)$. The symbol k is used to denote the different coils with $k = 1$ to K, where K is the total number of coils. The symbol g is used to denote different phase encoding gradients, and the value of g is from 1 to F, where F is the number of phase encoding gradients. This expression can be converted into the matrix form for each position x along the horizontal direction of the image, as shown in (3.13).

We can simplify (3.13) as:

$$A(x) = G(x) \times I(x), \quad , x = 1 \quad to \quad M, \tag{3.14}$$

where A(x), G(x), and I(x) represent the left, middle and right items in (3.13). Their dimensions are $KF \times 1$, $KF \times N$, and $N \times 1$. K is the number of coils, and F is the number of phase encoding gradients for each coil. M and N are the resolution of the reconstructed image. Typically M and N are 256 by 256 or 128 by 128.

Note that A(x) contains the F phase encoded values for all K coils. It is essentially a one-dimensional DFT of the chosen k-space data. Also, I(x) is an N-element vector representing one column of the image to be reconstructed and x is the horizontal coordinate of that column. G(x) can be constructed based on the sensitivity profiles and phase encodes used. If an image has M columns, then x ranges from 1 to M. For each particular x, we have an equation such as (3.14). These M equations can be constructed and solved independently of each other, which means each column of the image can be reconstructed independent of each other. Increasing M and N increases the computation load. It can also be seen that the Gain matrix G(x) becomes larger when K and F increase, thus increasing the computation load.

3.5 Loop Level Parallelization

In this section, the coarse grain parallelization of the image reconstruction is presented. As shown in the previous section, the SPACE RIP reconstruction algorithm is computed column by column. The algorithm begins by reading k-space data from the data file, then a 1D DFT is computed along the x direction, followed by a major loop reconstructing the columns one by one. This loop has M iterations, where M is the x dimension of the reconstructed image. Inside each iteration, a matrix $G(x)$, as in (3.13) is constructed. The pseudoinverse of this matrix is then computed, and one column of the image is finally reconstructed by multiplying the inverse matrix with the vector $A(X)$ as in (3.13). Timing profiling of the program for a typical data set shows that the major loop occupies about 98.79 % of the total execution time. Accordingly, this loop is the bottleneck to be parallelized.

Both Pthread and OpenMP versions at the loop level are implemented. The speedup result on a 12 CPUs Sunfire workstation are shown in section 3.9. On a shared-memory multiprocessor computer, all CPUs share the same main memory and can work on the same data concurrently. The major advantage of the shared-memory machine is that no explicit message-passing is needed, thus making it easier for programmers to parallelize the sequential code of an application compared to message-passing-based parallel languages, such as PVM or MPI.

Multithreaded programming is a programming paradigm tailored to shared-memory multiprocessor systems. Multithreaded programming offers an alternative to multi-process programming that is typically less demanding of system resources – here the collection of interacting tasks are implemented as multiple threads within a single process. The programmer can regard the individual threads as running concurrently and need not implement task switching explicitly, which is instead handled by the operating system or thread library in a manner similar to that for task switching between processes. Libraries and operating system support for multithreaded programming are available today on most platforms, including almost all available Unix variants. However, it is worth noting that there is a certain amount of overhead for handling multiple threads, so the performance gain archived by parallelization must outweigh this overhead. In our application, the loop level parallelizations are at the coarse grain level, thus justifying the overhead.

Pthread [31] is a standardized model for dividing a program into subtasks whose executions can be interleaved or run in parallel. The OpenMP Application Program Interface (API) [77] supports multi-platform shared-memory parallel programming in C/C++ and Fortran on almost all architectures. Additionally, it is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications.

It is worth noting that static variables are shared across all threads for both Pthread and OpenMP programming. In the SPACE RIP code, some CLAPACK [68] routines are used. The CLAPACK [68] routines, however, have many unnecessary static local variables, which are not thread-safe since they cause some unwanted sharing. If not dealt with, this unintended variable sharing causes false results or may affect performance.

In the current implementation, the memory for A(x), G(x) and I(x) as shown in (3.14) are pre-allocated. Thus the program structure is quite simple, as all the threads can work on independent memory locations and return the result to independent memory locations. No communication issue needs to be considered due to the problem property. In our implementation, a dynamic load balancing strategy is used for task distribution.

In fact, load balancing is not a big issue for our test platform because all the slave nodes have similar performance and task computation loads according to our observation.

An MPI version of the loop level parallelization is implemented on a Linux Cluster. The difference from the above SMP-based solution is that the MPI version needs explicit message-passing. Specifically, the M iterations in the loop are distributed to slave nodes dynamically. After the computation of the pseudoinverse for each column, the slave nodes send back the result (Pseudo inverse of the Gain matrix) to the master nodes. The master then sends a new column index to these slave nodes. Such a process continues until all iterations are completed. At the beginning, the master nodes send all necessary information to slaves, including the phase encoding gradient data and necessary information about the image, such as image dimension. Also at each iteration, the slave sends back $KF \times N$ double precision complex numbers as the result, which causes relatively heavy communication overhead.

3.6 Parallel SVD for Complex Matrices

The pseudoinverse of the gain matrix G(x) is solved by the singular value decomposition. In this section, we present the parallelization of the one-sided Jacobi SVD algorithm. The current existing algorithms for SVD are briefly reviewed first. Then a one-sided Jacobi update algorithm for complex matrices is proposed. This is important because the gain matrix is complex in this particular application. Then our parallel implementation is presented with the parallel ordering of GaoThomas [78]. GaoThomas parallel ordering is briefly reviewed and related implementation issues on SMP are discussed. The parallelization is implemented both on the current SMP and cellular architecture, the latter of which is under development. The speedup result is presented in Section 3.9.

3.6.1 Singular Value Decomposition

One of the important problems in mathematical science and engineering is singular value decomposition (SVD). The SVD forms the core of many algorithms in signal processing and has many interesting applications such as data compression, noise filtering, and image reconstruction in biomedical imaging. It is one of the most important factorizations of a real or complex matrices and is a computation-intensive problem. A SVD of real or complex m by n matrix is its factorization into the product of three matrices:

$$A = U \Sigma V^H, \tag{3.15}$$

where U is an m by n matrix with orthogonal columns, Σ is an n by n non-negative diagonal matrix, and V is an n by n orthogonal matrix. Here we use H to denote the complex conjugate transpose of a matrix. If a matrix is a real matrix, then H is the transpose operation.

There are many algorithms for solving the SVD problem. Firstly, the QR algorithm is used to solve singular value decomposition of a bidiagonal matrix. QR is used to compute singular vectors in LAPACK's [68] computational routine xBDSQR, which is used by the driver routine of xGESVD to compute the SVD of dense matrices. The xGESVD routine first reduces a matrix to bidiagonal form, and then calls the QR routine xBDSQR to find the SVD of the bidiagonal matrix. Originally, the SPACE RIP sequential code utilizes ZGESVD routine to solve the SVD problem of a complex matrix. It is worth noting that the Matlab SVD routine uses LAPACK routines DGESVD (for real matrices) and ZGESVD (for complex matrices) to compute the singular value decomposition.

Another approach is the divide-and-conquer algorithm. It divides the matrix into two halves, computes the SVD of each half, and integrates the solutions together by solving a rational equation. Divide-and-conquer is implemented in the LAPACK [68] routine xBDSDC, which is used by LAPACK driver routine xGESDD to compute the SVD of a dense matrix. It is currently the fastest routine available in LAPACK to solve the SVD problem of a bidiagonal matrix larger than about 25 by 25 [79]. xGESDD is currently the LAPACK algorithm of choice for the SVD of dense matrices. However, to our knowledge, there is no current parallel version of the ZGESVD routine or the ZGESDD routine in ScaLAPACK [80], which is a parallel version of LAPACK.

Finally, there is Jacobi's algorithm [81, 82]. It is most suitable for parallel computing. This transformation algorithm repeatedly multiplies on the right by elementary orthogonal matrices (Jacobi rotations) until it converges to $U\Sigma$, and the product of the Jacobi rotations is V. The Jacobi approach is slower than any of the above transformation methods, but has the useful property that it can deliver tiny singular values, and their singular vectors, much more accurately than any of the above methods, provided that it is properly implemented [83, 84]. Specifically, it is shown that the Jacobi algorithm is more accurate than the QR algorithm [85].

3.6.2 One-sided Jacobi Algorithm

In our implementation, we focus on the one-sided Jacobi SVD algorithm since it is most suitable for parallel computing. In the one-sided Jacobi algorithm, in order to compute an SVD of an $m \times n$ matrix A, most algorithms adopt Jacobi rotations. The idea is to generate an orthogonal matrix V such that the transformed matrix $AV = W$ has orthogonal columns. Normalizing the Euclidean length of each nonnull column of W to unity yields:

$$W = U\Sigma, \tag{3.16}$$

where the U is a matrix whose nonnull columns form an orthonormal set of vectors and Σ is a nonnegative diagonal matrix. Since $V^H V = I$, where I is the identity matrix, we have the SVD of A given by $A = U\Sigma V^H$.

Hestenes [86] uses plane rotations to construct V. The remainder of this subsection first reviews Hestenes's algorithm for real matrices and then extends the algorithm for complex matrices.

Hestene generates a sequence of matrices $\{A_k\}$ using the rotation

$$A_{k+1} = A_k Q_k, \tag{3.17}$$

where the initial $A_1 = A$ and Q_k is a plane rotation. Let $A_k \equiv (\vec{a}_1^{(k)}, \vec{a}_2^{(k)}, \cdots, \vec{a}_n^{(k)})$, and $Q_k \equiv q_{rs}^{(k)}$. Suppose Q_k represents a plane rotation in the (i,j) plane, with $i < j$, Let us define:

$$\begin{aligned} q_{ii}^{(k)} &= c, & q_{ij}^{(k)} &= s, \\ q_{ji}^{(k)} &= -s, & q_{jj}^{(k)} &= c. \end{aligned} \tag{3.18}$$

The postmultiplication by Q_k affects only two columns:

$$(\vec{a}_i^{(k+1)}, \vec{a}_j^{(k+1)}) = (\vec{a}_i^{(k)}, \vec{a}_j^{(k)}) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \tag{3.19}$$

To simplify the notation, let us define:

$$\begin{aligned} \vec{u}' &\equiv \vec{a}_i^{(k+1)}, & \vec{u} &\equiv \vec{a}_i^{(k)}, \\ \vec{v}' &\equiv \vec{a}_j^{(k+1)}, & \vec{v} &\equiv \vec{a}_j^{(k)}. \end{aligned} \tag{3.20}$$

Then we have:

$$(\vec{u}', \vec{v}') = (\vec{u}, \vec{v}) \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \tag{3.21}$$

For real matrices, to make the two new columns orthogonal, we have to satisfy $(\vec{u}')^T \vec{v}' = 0$. Further mathematical manipulations yield:

$$(c^2 - s^2)w + cs(x - y) = 0, \tag{3.22}$$

where $w = \vec{u}^T \vec{v}$, $x = \vec{u}^T \vec{u}$, $y = \vec{v}^T \vec{v}$.

Rutishauser[87] proposed the formulas as in (3.23) to solve (3.22). They are in use because they can diminish the accumulation of rounding errors:

$$\begin{aligned} \alpha &= \frac{y-x}{2w}, & \tau &= \frac{sign(\alpha)}{|\alpha| + \sqrt{1+\alpha^2}}, \\ c &= \frac{1}{\sqrt{1+\tau^2}}, & s &= \tau c. \end{aligned} \tag{3.23}$$

We set $c = 1$ and $s = 0$ if $w = 0$.

3.6.3 Extension to Complex Matrices

It is noteworthy that the above formulas only apply to real matrices. In order to make the two new columns orthogonal In the case of complex matrices, we have to make $(\vec{u}')^H\vec{v}' = 0$. This still yield (3.22), except that the inner products w, x and y are now defined as:

$$w = \vec{u}^H\vec{v}, x = \vec{u}^H\vec{u}, y = \vec{v}^H\vec{v}. \tag{3.24}$$

The x and y variables are still real numbers, but w may be complex number, which makes the solution, as shown in (3.23) no longer valid.

Park [88] proposed a real algorithm for Hermitian Eigenvalue decomposition for complex matrices. Henrici [89] proposed a Jacobi algorithm for computing the principal values of a complex matrix. Both use two sided rotations. Inspired by their algorithms, we derived the following one sided Jacobi rotation algorithm for complex matrices. We modify the rotation as follows:

$$(\vec{u}', \vec{v}') = (\vec{u}, \vec{v}) \begin{pmatrix} e^{j\beta} & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} e^{-j\beta} & 0 \\ 0 & 1 \end{pmatrix}, \tag{3.25}$$

where we get the angle β from w: $w = |w|e^{j\beta}$. The formula to get c and s are as follows:

$$\begin{array}{llll} \alpha & = \frac{y-x}{2|w|}, & \tau & = \frac{sign(\alpha)}{|\alpha|+\sqrt{1+\alpha^2}} \\ c & = \frac{1}{\sqrt{1+\tau^2}}, & s & = \tau c. \end{array} \tag{3.26}$$

We set $c = 1$ and $s = 0$ if $|w| = 0$.

The idea is to first apply the complex rotation shown in (3.25). After this complex rotation, the inner product of the two updated columns becomes real number. It is easy to verify that the $(\vec{u}')^H\vec{v}' = 0$ is satisfied with our proposed rotation algorithm.

If the matrix V is desired, the plane rotations can be accumulated. We compute

$$V_{k+1} = V_k Q_k \tag{3.27}$$

and update the A and V simultaneously.

```
Rotation_of_two_column(colu, colv)
{

     /*colu and colv are two
     columns of complex numbers */
     /*The lengh of column is n*/

     w=inner_product(colu,colv);

     if(|w| <= delta)
     {
       converged <- true;
       return ;
     }
     else converged <- false;

     x=inner_product(colu, colu);
     y=inner_product(colv, colv);

     computer rotation parameter c,s
     from w, x, y according to
     Equation 3.26;

     update colu, colv according
     to rotation Equation 3.25;
}
```

Listing 3.1: Rotation of two columns of complex numbers

The pseudo-code of a Jacobi routine for complex matrices is shown in Listing 3.1. We refer to the algorithm in Listing 3.1 as the “basic rotation routine”. To simplify the case, the V matrix updating is not included in this kernel.

3.6.4 Parallel Scheme

The plane rotations have to be applied to all column pairs exactly once in any sequence (a sweep) of $n(n-1)/2$ rotations. Several sweeps are required so that the matrix converges. A simple sweep can be a cyclic-by-rows ordering. For instance, let us consider a matrix with 4 columns. With the cyclic-by-rows order, the sequence of a sweep is:

$$(1,2),(1,3),(1,4),(2,3),(2,4),(3,4). \tag{3.28}$$

It is easy to see that some pairs are independent and may be executed in parallel if we change the order in the sequence. Another possible sequence for a sweep can group independent pairs and execute them in parallel:

$$\{(1,2),(3,4)\},\{(1,4),(2,3)\},\{(1,3),(2,4)\}, \tag{3.29}$$

where the pairs in curly brackets are independent.We call each of these groups a step. This feature motivates the proposal of many parallel Jacobi ordering algorithms [78, 90–92] in which the $n(n-1)/2$ rotations required to complete a sweep are organized into groups of independent transformations. Gao and Thomas's algorithm [78] is optimal in terms of achieving both the maximum concurrency in computation and minimum overhead in communication.

We implemented the Gao and Thomas algorithm. This algorithm computes the pairs of n elements on $n/2$ processors when n is a power of 2. In each computation step, each processor computes one pair. During the communication stage, each processor exchanges only one column with another processor. The total number of computation steps is $(n-1)$. The detailed recursive divide and exchange algorithm is explained in [78]. We only give one example of parallel ordering in Table 3.2 for a matrix with 8 columns.

Table 3.2: Parallel ordering of GaoThomas algorithm

step 1	(1, 2)	(3, 4)	(5, 6)	(7, 8)
step 2	(1, 4)	(3, 2)	(5, 8)	(7, 6)
step 3	(1, 8)	(3, 6)	(5, 4)	(7, 2)
step 4	(1, 6)	(3, 8)	(5, 2)	(7, 4)
step 5	(1, 5)	(3, 7)	(6, 2)	(8, 4)
step 6	(1, 7)	(3, 5)	(6, 4)	(8, 2)
step 7	(1, 3)	(7, 5)	(6, 8)	(4, 2)

In our shared memory implementation, the number of slave threads p can be set to be the number of available processors. All the column pairs in one step can be treated as a work pool. The works in this work pool will be distributed to the p slave threads, where

$1 \leq p \leq \frac{n}{2}$. After each step, we implemented a barrier to make sure the step $k+1$ always uses the updated column pairs from step k. At the end of each sweep, we check whether the convergence condition is satisfied. If not, we start a new sweep again. Otherwise, the program terminates.

The convergence behavior of different orderings may not be the same. Hansen [93] discusses the convergence properties associated with various ordering. In our implementation, we chose to use a threshold approach in order to enforce convergence [94]. We omit any rotation if the inner product $(\vec{u})^H \vec{v}$ of the current column pairs $\vec{u}$ and $\vec{v}$ is below a certain threshold δ. The δ is defined as :

$$\delta = \epsilon \cdot \sum_{i=1}^{N} A[i]^H A[i], \tag{3.30}$$

where ϵ is the machine precision epsilon and $A[i]$ is the ith column of the initial A matrix. At the end of each sweep, if all the possible pairs in the sweep have converged according to the above standard, then the problem has converged.

3.6.5 Group-based GaoThomas Algorithm

As stated previously, the GaoThomas algorithm can compute $n(n-1)/2$ rotations of a matrix with n columns on $n/2$ processors. When the size of the matrix increases, group-based GaoThomas can be adopted. For instance, when the matrix size is $2n$ and we only have $n/2$ processors, we can group two columns together and treat them as one single unit. Then the primary algorithm for a matrix with n columns can be used.

For a matrix with n columns, if we group g columns together as a group, then we have n/g groups and can use the basic GaoThomas algorithm for n/g elements, except each element is a group. For instance, operations on a matrix 16 by 16 can set the group size to be 2, yielding 8 groups for which we can still use the divide and exchange algorithm shown in Table 3.2. The only difference is that each bracket in the table is a rotation of two groups, each group containing 2 columns in this case.

```
Rotation_of_two_group(group_a, group_b)
{

 /*group size is g */
 /*group_a contains columns $u_i, i = 1, g$*/
 /*group_b contains columns $v_i, i = 1, g$*/

 if(current step is step 1)
 {
    for i=1 to g
       for j=i+1 to g
         Rotate_of_two_column($u_i, u_j$);

    for i=1 to g
       for j=i+1 to g
         Rotate_of_two_column($v_i, v_j$);
 }

 for i=1 to g
       for j=1 to g
         Rotate_of_two_column($u_i, v_j$);

}
```

Listing 3.2: Rotation of two groups

Therefore, in the group-based algorithm for a matrix with n columns and a group size g, one sweep contains $n/g - 1$ steps, and each step contains $n/2g$ instances of a rotation of two groups, which can run in parallel on a maximum of $n/2g$ processors. The pseudo-code for rotating two groups is shown in Listing 3.2. It is easy to find out that after one sweep, all $n(n-1)/2$ basic rotations of two columns are computed.

3.7 Optimization of Memory Access

This section discusses several memory access approaches that can be integrated into the rotation routines shown in Listings 3.1 and 3.2.

3.7.1 Naive Approach

The default memory allocation using “malloc()” in the Cyclops-64 simulator is from the offchip memory, while the local variables are allocated from the stack located on the onchip scratch-pad memory. Assuming that the matrix data originally reside on the offchip memory, we implemented an SVD program where all the memory accesses are from the offchip memory. This implementation is referred to as the naive version in the following discussions. Also, the loop within the inner product computation in the rotation routine is implemented without any loop unrolling in this version.

3.7.2 Preloading

In order to reduce the cycles spent on memory accesses, we can preload the data from the offchip memory to the onchip scratch-pad memory. Thus the data accesses in the computation part of the rotation routine are directly from the onchip memory. The updated data are then stored back to the offchip memory.

There are two ways to preload data. The simplest way is to use the “memcpy” function from the C library. The pseudo-code for the “memcpy” preloading in the two-column rotation routine is shown in Listing 3.3. We refer to the code segment from line 10 to line 12 as the “computation core”, which consists of the computation of three inner products and a column rotation. Preloading for the group-based rotation routine is similar, except that two “groups” of columns are preloaded. The “memcpy”-based preloading has the problem of paying extra overhead of function calling. Additionally, the assembly code of the “memcpy” function is not fully optimized, which is shown with analysis in the next section. To overcome these two problems, we implement preloading by using an optimized inline assembly code instead of a function call. We refer to this approach as the “inline” approach. For this approach, each “memcpy” function call is replaced with a segment of inline assembly code. The assembly code segments for the “memcpy” and “inline” preloading approaches (either the group-based rotation routine or the basic rotation routine) are shown in Listing 3.6 and Listing 3.7. From the listings, we can see that memcpy and inline approaches have different instruction scheduling. The former conducts one “LDD” instruction followed by one “STD” and repeats for a sufficient number of times until all the data are moved successfully. The latter, in contrast, issues several “LDD” instructions in a row (in our case, 8 LDDs in a row) followed by several “STD”s in a row. The effect of different ways of instruction scheduling on the total memory access cycles is analyzed in Section 3.8.

```
Rotation_of_two_column(colu, colv)
{

  Allocate local_colu, local_colv
  on the scratch-pad;

  memcpy (local_colu <-colu);
  memcpy (local_colv <-colv);

  conduct three inner products and
  column rotation on local_colu, local_colv
  as in Listing.3.1

  memcpy (colu <-local_colu);
  memcpy (colv <-local_colv);
}
```

Listing 3.3: Basic rotation routine with preloading using "memcpy"

3.7.3 Loop Unrolling of Inner Product Computation

There are three inner product function calls in the rotation routine. We implemented two versions of loop unrolling for the loop in the inner product computation: unrolling the loop body 4 times or 8 times. The idea is that loop unrolling makes it possible to schedule instructions from multiple iterations, thus facilitating the exploitation of instruction level parallelism.

3.8 Performance Model

In this section, the performance model to dissect the execution cycles is introduced first. This model is then applied to analyze and compare the cycles spent on memory accesses for the memory access approaches discussed in the previous section.

3.8.1 Dissection of Execution Cycles

We begin with a simple execution trace example in Listing 3.4 to illustrate how to dissect total execution cycles into several parts. In the listing, the first column is the current cycle number. We notice that at cycle 98472, there is a note “DLL = 1”, which means that there is a one-cycle latency related to memory access. The reason for this latency is that at cycle 98472 the instruction needs the operand R9, which is not ready at cycle 98472 because the LDD instruction at cycle 98470 has two cycles of latency. Similarly, at cycle 98475, the FMULD instruction needs the input operand R8 generated by the FDIVD instruction at cycle 98469. R8 is not ready at cycle 98475 and needs an extra latency of 25 cycles since the FDIVD instruction has 30 cycles of latency from the float point unit. Counting the total number of cycles from cycle 98469 till cycle 98501, there are 33 cycles which include 7 instructions, 1 cycle of “DLL” and 25 cycles of “DLF”.

The integer unit may also cause certain latency called “DLI”, which is similar to the “DLF” in the trace example. Therefore, we have the following equation:

$$\begin{aligned} Total \quad cycles = & \quad INST \\ & + \quad DLL + \quad DLF + \quad DLI, \end{aligned} \tag{3.31}$$

where the “ INST” part stands for the total number of instructions, “DLL” represents the cycles spent on memory access, “DLF” represents the latency cycles related to floating point instructions, and “DLI” represents the latency cycles related to integer instructions. When we change from the naive approach to the previously discussed memory access schemes, the “DLL” part is the most affected part. Our goal is to reduce this part by using preloading or loop unrolling. The “INST” part is also affected because the “memcpy” or “inline” approach incurs extra instructions. The “DLF” and “DLI” part are approximately unchanged because they are related to either the floating or integer point unit computation that does not change with the change of memory access schemes. The next section gives an estimate of the gain and cost in terms of “DLL” and “INST” for different approaches.

```
98469  FDIVD     R8,R60,R8
98470  LDD       R9,R3,96
98471  ORI       R21,R0,0
98472  FDIVD     R20,R9,R62       DLL = 1
98474  LDD       R60,R3,104
98475  FMULD     R6,R61,R8        DLF = 25
98501  STD       R8,R3,160
```

Listing 3.4: Example of dissection of exectution cycles

3.8.2 Analysis of Naive Approach

All memory accesses in the naive approach are from the offchip memory, and the computation core part has a large number of "DLL" latency cycles. We denote the size of the matrix as $n \times n$. Each element of this matrix is a double complex number. We focus on one sweep that consists of $\binom{n}{2}$ basic rotations for either the non-group-based approach or the group-based approach. A basic rotation, as shown in Listing 3.1 consists of two different parts, the inner product part and the column rotation part. We analyze the total "DLL" latency cycles for both of them in this subsection.

First, there are three inner product function calls in the basic rotation routine. Each one of them consists of n iterations, each iteration producing a multiplication of two complex numbers and adding it to the sum. The execution trace of the innermost iteration is shown in Listing 3.5. In this example, the offchip latency is set to be 80 cycles. From the trace, we see that the innermost iteration has a "DLL = 76". In general, if we preset the offchip latency to be L cycles, then the total number of "DLL" cycles in each iteration is $L - 4$. Therefore, in one sweep, the total number of "DLL" cycles within the inner product part is:

$$DLL_{innerproduct} = \binom{n}{2} \times 3 \times n \times (L-4), \tag{3.32}$$

where "3" means that the inner product function is called three times inside one basic rotation routine, n is the number of iterations in the inner product function, and $L - 4$ is the number of "DLL" cycles within the innermost iteration.

```
105069   SHLI     R19,R12,4
105070   ADD      R18,R19,R58
105071   ADD      R7,R19,R57
105072   LDD      R9,R7,8
105073   LDD      R17,R18,8
105074   LDD      R10,R18,0
105075   LDD      R11,R7,0
105076   ADDI     R12,R12,1
105077   CMPLT    R15,R12,R61
105078   FMULD    R16,R9,R17     DLL = 76
105155   FMULD    R8,R9,R10
105156   FMAD     R16,R11,R10    DLF = 4
105161   FSUBD    R8,R0,R8
105162   FMAD     R8,R11,R17     DLF = 5
105168   FADDD    R14,R14,R16    DLF = 3
105172   FADDD    R13,R13,R8     DLF = 6
105179   BNE      R15,−64
```

Listing 3.5: Trace of the innermost iteration in the inner product routine

Second, for the column rotation part in the basic rotation routine, we conduct a similar analysis. The total number of "DLL" cycles of this part is:

$$DLL_{column\_rotation} = \binom{n}{2} \times n \times (L - 4). \tag{3.33}$$

Therefore the total number of "DLL" cycles in the naive implementation of GaoThomas algorithm (either the group-based or the non-group-based) in one sweep is:

$$\begin{aligned} DLL_{naive} &= DLL_{innerproduct} + DLL_{column\_rotation} \\ &= \tbinom{n}{2} \times n \times (4L - 16). \end{aligned} \tag{3.34}$$

3.8.3 Analysis of "Memcpy" Approach

Using either the "memcpy" or "inline" preloading approach, the computation core accesses data from the onchip memory. The "DLL" part in the computation core is approximately zero due to the overlap of the short onchip memory access latency (2 cycles) with the float point unit latency. Therefore, from the program without preloading to the program with preloading, the decrease of the total number of "DLL" cycles in the computation core is DLL_{naive}, which is the cycles we save by using preloading, and thus the gain we expect to get.

Moving data from the offchip memory to the onchip memory results in an extra cost, which consists of two parts: the first part is the total "DLL" cycles in the code segment that is responsible for moving data, and the second part is the extra instructions incurred.

Firstly, we derive the total number of "memcpy" function calls (which are responsible for loading data "in"). For the basic non-group-based GaoThomas algorithm, there is a total of $\binom{n}{2}$ basic rotations (shown in Listing 3.1) in one sweep. A basic rotation needs to load in two columns, each of length n. Loading a double complex number needs two "LDD" instructions. Therefore, the total number of "LDD"s for preloading data is:

$$\begin{aligned} LDD_{memcpy\_no\_group} &= \binom{n}{2} \times 2 \times n \times 2 \\ &= \binom{n}{2} \times 4n, \end{aligned} \tag{3.35}$$

where the first "2" stands for loading "two" columns, n is the length of the column, and the second "2" means that loading a double complex number needs two LDDs.

For the group-based algorithm, if the group size is g, there is a total of $\binom{n/g}{2}$ group-based rotations. At the beginning of each group-based rotation, we need to load in two groups of columns (i.e, $2 \times g$ columns) and each column needs $n \times 2$ LDDs. Therefore, the total number of LDDs for preloading data during one sweep is:

$$\begin{aligned} LDD_{memcpy} &= \binom{n/g}{2} \times 2g \times n \times 2 \\ &= \binom{n/g}{2} \times g \times 4n. \end{aligned} \tag{3.36}$$

If we treat the non-group-based GaoThomas algorithm as a group-based algorithm with group size one, then we can use (3.36) for either the group-based algorithm or non-group-based algorithm.

Secondly, we compute the latency incurred by the LDDs. The execution trace segment of the assembly code for the "memcpy" function is shown in Listing 3.6, with the offchip latency set to be 80. From Listing 3.6, we observe that each LDD instruction causes a long latency of 80 cycles. This latency is reflected where the "STD" instructions

```
105375  LDD     R6,R9,0
105376  STD     R6,R7,0      DLL = 80
105457  ADDI    R9,R9,8
105458  ADDI    R7,R7,8
105459  LDD     R6,R9,0
105460  STD     R6,R7,0      DLL = 80
105541  ADDI    R9,R9,8
105542  ADDI    R7,R7,8
105543  LDD     R6,R9,0
105544  STD     R6,R7,0      DLL = 80
105625  ADDI    R9,R9,8
105626  ADDI    R7,R7,8
105627  LDD     R6,R9,0
105628  STD     R6,R7,0      DLL = 80
```

Listing 3.6: Trace of the memcpy code segment

exist. If we preset the offchip latency to be L, then each "LDD" causes a latency of L cycles. So the total number of "DLL" cycles for preloading data using "memcpy" is:

$$\begin{aligned} DLL_{memcpy} &= LDD_{memcpy} \times L \\ &= \binom{n/g}{2} \times g \times 4n \times L. \end{aligned} \tag{3.37}$$

In summary, from the naive approach to the "memcpy"-based preloading approach, the extra cost paid is the DLL_{memcpy} while the cycles saved is DLL_{naive} as in (3.34). The preloading approach is a good choice whenever the cost is smaller than the gain.

In addition to the change in the total number of "DLL"s, we also observe the increase in the total instruction count as:

$$Total \quad INST \quad increase = \binom{n/g}{2} \times g \times 4n \times 2 \times 2, \tag{3.38}$$

where the first part $\binom{n/g}{2} \times g \times 4n$ is the total number of "LDD"s for preloading data. We need a same amount of "STD", thus a multiplication by 2. Also we need to use "LDD" and "STD" to store data back, thus another multiplication by 2.

3.8.4 Analysis of "Inline" Approach

The total amount of data preloaded for the "inline" preloading approach is the same as the "memcpy" approach. Therefore, the total number of "LDD"s of the inline approach is the same as the "memcpy" approach:

$$\begin{aligned} LDD_{inline} &= \binom{n/g}{2} \times 2g \times n \times 2 \\ &= \binom{n/g}{2} \times g \times 4n, \end{aligned} \tag{3.39}$$

where n is the matrix size and g is the group size..

The difference between the "inline" approach and "memcpy" approach is the scheduling of the LDD and STD instructions in the assembly code. As shown in Listing 3.6, each LDD in the "memcpy" approach is followed immediately by one STD. In the "inline" approach, 8 LDDs in a row are followed by 8 STDs in a row, as shown in Listing 3.7. From the trace we can see that we will have one "DLL=73" every 8 LDDs if we preset the offchip latency to be 80. If the offchip latency is L cycles, there is a "DLL=$L-7$" every 8 "LDD" instructions. Therefore, the total number of "DLL" cycles for preloading data using the "inline" approach is:

$$\begin{aligned} DLL_{inline} &= LDD_{inline}/8 \times (L-7) \\ &= \frac{1}{8} \times \binom{n/g}{2} \times g \times 4n \times (L-7). \end{aligned} \tag{3.40}$$

From (3.40), we can see very clearly that preloading data using the "inline" approach is better than using the "memcpy" approach because DLL_{inline} is approximately $1/8$ of DLL_{memcpy}.

From the naive approach to the "inline" preloading approach, the extra cost paid is the DLL_{inline}, while the cycles saved is DLL_{naive} as in (3.34). Increase in the total instruction count is computed according to (3.38). It is also noteworthy that the "inline" approach should have a smaller instruction count increase than the "memcpy" approach since the former does not need the instructions involved in function calling.

```
112688  LDD     R16,R9,0
112689  LDD     R17,R9,8
112690  LDD     R18,R9,16
112691  LDD     R19,R9,24
112692  LDD     R20,R9,32
112693  LDD     R21,R9,40
112694  LDD     R22,R9,48
112695  LDD     R28,R9,56
112696  STD     R16,R6,0         DLL = 73
112770  STD     R17,R6,8
112771  STD     R18,R6,16
112772  STD     R19,R6,24
112773  STD     R20,R6,32
112774  STD     R21,R6,40
112775  STD     R22,R6,48
112776  STD     R28,R6,56
```

Listing 3.7: Trace of the "inline" approach

3.8.5 Analysis of Loop Unrolling Approach

Loop unrolling only affects the inner product routine. For unrolling 4 times, each inner product routine now contains $n/4$ iterations, each one of them computing the sum of 4 multiplications of complex numbers. The trace of one iteration is shown in Listing 3.8, with the offchip latency preset to be 80. The trace shows that the innermost iteration consists of a "DLL = 72". In general, if we preset the offchip latency to be L cycles, then the "DLL" in each iteration is $L-8$ cycles. Therefore, in one sweep, the total number of "DLL" cycles incurred inside the inner product part is:

$$DLL_{innerproduct\_unroll4} = \binom{n}{2} \times 3 \times \frac{n}{4} \times (L-8). \tag{3.41}$$

A similar analysis of unrolling 8 times yields:

$$DLL_{innerproduct\_unroll8} = \binom{n}{2} \times 3 \times \frac{n}{8} \times (L-13), \tag{3.42}$$

where "$L-13$" comes from the trace of loop unrolling 8 times.

The difference between the naive and loop unrolling approaches is only in the inner product routine called by the basic rotation routine, while the column rotation and the update part in the basic rotation routine are kept unchanged. Therefore, from the naive approach to the loop unrolling approach, the total cycles saved is $DLL_{innerproduct} - DLL_{innerproduct\_unroll4}$ for unrolling 4 times, or $DLL_{innerproduct} - DLL_{innerproduct\_unroll8}$ for unrolling 8 times.

```
95288  SHLI    R14,R62,4
95289  ADD     R6,R14,R56
95290  ADD     R7,R14,R55
95291  LDD     R9,R7,8
95292  LDD     R10,R6,8
95293  LDD     R17,R6,16
95294  LDD     R18,R6,24
95295  LDD     R20,R6,32
95296  LDD     R21,R6,40
95297  LDD     R60,R6,56
95298  LDD     R15,R6,48
95299  LDD     R61,R6,0
95300  LDD     R11,R7,0
95301  LDD     R16,R7,16
95302  LDD     R12,R7,24
95303  LDD     R19,R7,32
95304  LDD     R13,R7,40
95305  LDD     R14,R7,48
95306  LDD     R8,R7,56
95307  ADDI    R62,R62,4
95308  FMULD   R6,R9,R61      DLL = 72
95381  FMULD   R9,R9,R10
95382  FSUBD   R6,R0,R6       DLF = 4
95387  FMAD    R6,R11,R10     DLF = 5
95393  CMPLT   R7,R62,R57
95394  FMAD    R9,R11,R61
95395  FMAD    R6,R16,R18     DLF = 8
95404  FMAD    R9,R16,R17     DLF = 1
95406  FMSD    R6,R12,R17     DLF = 8
95415  FMAD    R9,R12,R18     DLF = 1
95417  FMAD    R6,R19,R21     DLF = 8
95426  FMAD    R9,R19,R20     DLF = 1
95428  FMSD    R6,R13,R20     DLF = 8
95437  FMAD    R9,R13,R21     DLF = 1
95439  FMAD    R6,R14,R60     DLF = 8
95448  FMAD    R9,R14,R15     DLF = 1
95450  FMSD    R6,R8,R15      DLF = 8
95459  FMAD    R9,R8,R60      DLF = 1
95461  FADDD   R59,R59,R6     DLF = 8
95470  FADDD   R58,R58,R9     DLF = 1
95472  BNE     R7,-160
```

Listing 3.8: Trace of one iteration of the inner product routine unrolled 4 times

3.9 Experimental Results

In this section, we present experimental results: the speedup of the loop level parallelization and the fine level parallelization of SVD, as well as the comparison of different memory access schemes.

3.9.1 Target Platform and Simulation Environment

Our test platforms include a Sunfire SMP machine from Sun Inc., a Linux cluster and a cellular computer architecture Cyclops-64.

The software tool chain of the Cyclops-64 platform currently provides a compiler, linker and simulator for users. A number of optimization levels are supported by the compiler. A functional accurate simulator (FAST) is also provided. The main features are: (1) the simulator supports most of the features of Cyclops-64 architecture, including multithreaded execution, Cyclops-64 ISA, floating point unit, interrupts, memory mapped features (interthread interrupt and wakeup signal); (2) FAST can generate the execution trace and/or an instruction statistics report to help a software/application developer tuning and optimizing a program. Furthermore, the macros “Trace On” and “Trace off” allow us to generate trace for a specific segment of code and save the trace to a file; (3) It can also generate a timing result at the cycle level of a program. For an application to be simulated, the code must be slightly modified. The Cyclops-64 software tool chain runs on Linux environment.

We also developed a Trace Analyzer that can take the output trace from the simulator and generate the dissection of execution cycles and analysis of the code simulated. The trace analyzer can generate the following statistics: (1) the dissection of the execution cycles to the four different parts, as shown in (3.31); (2) the analyzer can also generate statistics about the total “DLL” related to a certain instruction. For instance, in the example shown in Listing 3.7, the “DLL” latencies caused by the “LDD” instruction are reflected in the STD instruction. We call such latencies “associated” with the STD instruction. The analyzer can sum the total “DLL”s associated with the STD instruction. The sum is the total memory access cycles in the code segment of data preloading, as in (3.37) or (3.40). Similarly, the “DLL”s associated with the float point instructions such as “FMULD” or “FMAD” are the “DLL”s in the computation core, as in (3.34).

3.9.2 Loop Level Parallelization

The loop level parallelization is carried out on a Sunfire SMP machine and a Linux cluster. The SMP machine used is the DBI-RNA1 at the Delaware Biotechnology Institute. DBI-RNA1 is a Sun Sunfire 4800 Server with 12 SPARC 750MHz CPUs, and a 24 gigabyte memory. The code has also been ported to the Cellular computer architecture Cyclops-64. However, due to the fact that the Simulator at this stage is slow to carry out the loop level parallelization experiment, we only present the Loop level parallelization result on a Sunfire SMP machine and a Linux cluster.

In the data used in this experiment, the number of coils is 4, the image size is 128 by 128 and the number of phase encodes is 38.

Fig. 3.10 presents the result of both Pthread and OpenMP. The speedup of both the total execution time and the loop only are presented. From the figure, it is seen that both Pthread and OpenMP achieved near linear performance up to 12 threads. This is due to the fact that the tasks (iterations) of the loop are totally independent of each other.

According to the well known “Amdahl’s” law, if a program can be expressed as two portions, the serial portion S and the parallel portion P, then the time $T(n)$ required to complete a task on n parallel processors is:

$$T(n) = S + \frac{P}{n}, \tag{3.43}$$

and the speedup for n CPUs can be expressed as:

$$sp = \frac{T(1)}{T(n)} = \frac{S + P}{S + \frac{P}{n}}. \tag{3.44}$$

From the above equation, it is seen that the speedup of a parallel program cannot continue to grow forever. Instead, there is a theoretical limit for the speedup:

$$sp_{\infty} = \lim_{n \to \infty} sp = \frac{S + P}{S}. \tag{3.45}$$

According to our timing experiment, the total execution time of the loop body occupies about 98.79 % of the total execution of the sequential program, which means the limit of the speedup is approximately 82.64. We assume that for bigger data sets, the loop body time percentage will be even bigger, and in real application, the loop body may be used to handle real-time streams. So we focus only on the speedup of the loop itself in the following discussion. For instance, in the fine level parallelization part, only the speedup of the loop is shown.

A MPI version is also implemented and tested on a Linux Cluster. The Linux cluster "Comet" consists of 18 nodes, each containing two 1.4 GHz AMD Athlon processors and 512MB of DDR SDRAM memory. The interconnection network for the nodes is a switched 100Mbps ethernet. From Fig. 3.11, it can be seen that the MPI version achieves a good speedup until the number of slave nodes reaches around 20. The speedup gradually stops increasing when the number of slave nodes is greater than 20. The reason is that when the number of slaves increases, the work load distributed to each slave becomes smaller, which does not justify the communication overhead at the initialization stage.

3.9.3 Fine Level Parallelization: Parallel SVD on SMP Machine Sunfire

In this section, the speedup result of the one-sided Jacobi SVD on Cyclops-64 for complex matrices is presented. Fig. 3.12 shows the speedup for the matrix sizes 128 by 128 through 1024 by 1024 (Pthread version). The numbers in the matrix are uniformly random double precision complex numbers. From the figure, it can be seen that for a small problem size such as 128 by 128, the speedup is limited to approximately 4. The reason is that the task grain is not big enough to justify the overhead associated with the thread creation and synchronization such as barrier and mutex. In order to achieve a

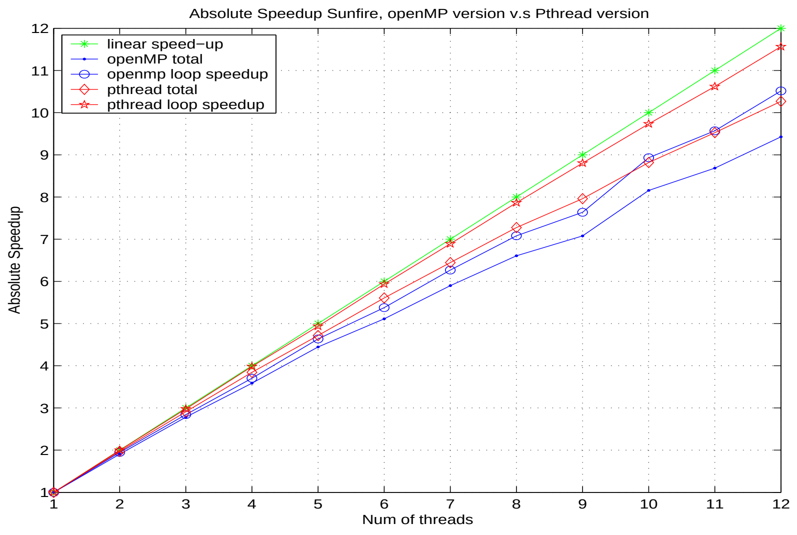


Figure 3.10: Speedup of loop level parallelization on Sunfire

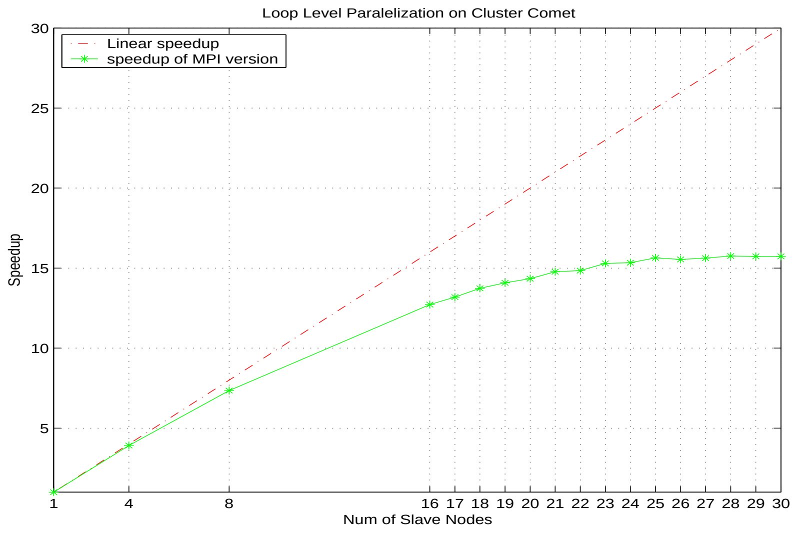


Figure 3.11: Speedup of loop level parallelization on Linux cluster

good speedup for small problem size, small thread synchronization overhead is necessary, which is a good feature of Cyclops-64 architecture.

3.9.4 Fine Level Parallelization: Parallel SVD on Cyclops-64

In this section, the speedup result of the one-sided Jacobi SVD on Cyclops-64 for complex matrices is reported. Fig. 3.13 shows the speedup for the matrix sizes 128 by 128, 64 by 64 and 32 by 32. The numbers in the matrix are uniformly random double precision complex numbers. According to GaoThomas parallel ordering, the maximum speedup for a n by n matrix is $\frac{n}{2}$. In our experiment, for the matrix size 128 by 128, the measured speedup is 43, which is approximately 68% of the theocratical value.

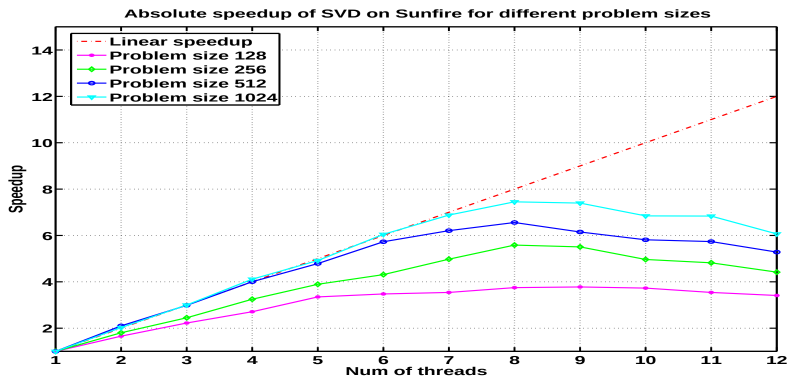


Figure 3.12: Speedup of parallel one-sided Jacobi complex SVD on Sunfire

In Fig. 3.14, we compare the performance of the complex SVD on Sunfire and Cyclops-64. From the figure, it can be seen that Cyclops-64 shows much better performance for matrix size 128. The actual biomedical data shows a similar result and is not plotted due to the space limitation.

It is worth noting that Jacobi SVD is slower than other SVD algorithms. For the data with a matrix size 152 by 128, our implementation is about 2 times slower than ZGESVD in the CLAPACK package, which means, with more 2 processors, the parallel SVD is better than ZGESVD.

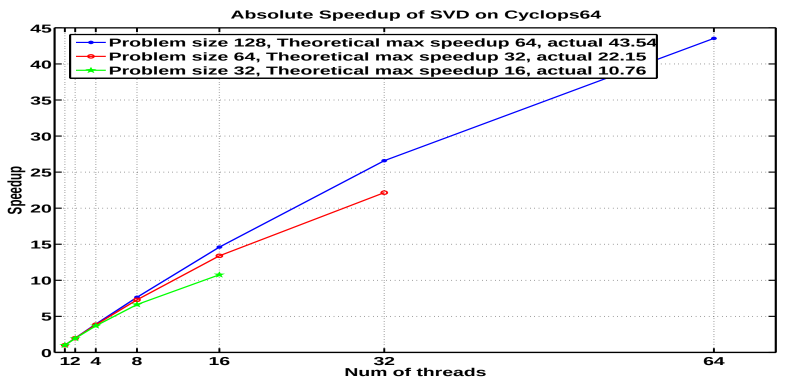


Figure 3.13: Speedup of parallel one-sided Jacobi complex SVD on Cyclops-64

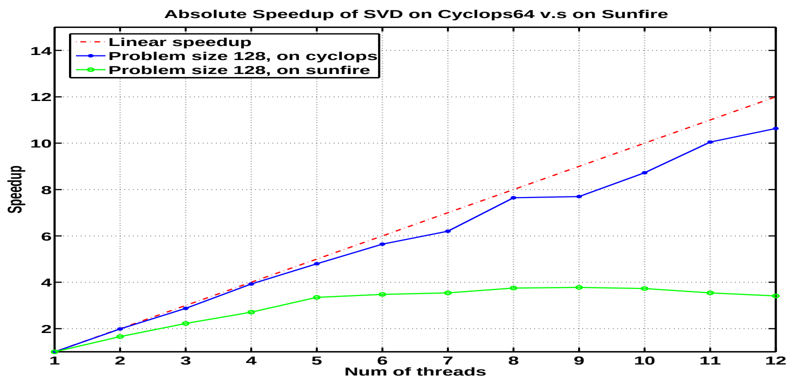


Figure 3.14: Parallel SVD on Cyclops-64 versus Sunfire

3.9.5 Simulation Results for Different Memory Access Schemes

In this subsection, the simulation results of the SVD GaoThomas algorithm are presented for problem size $n = 32$ and $n = 64$. The default configuration of offchip latency is 36 cycles. If there is a heavy load of memory access operations and memory access contention from different threads, the effective offchip latency becomes larger. Therefore, simulation results for the offchip latency of 80 cycles are also presented. The simulation environment is introduced first, then a data table is presented to show the change of the "DLL" part in different versions of implementation, including the naive approach, preloading using "memcpy" or "inline", and loop unrolling 4 times or 8 times.

The actual numbers measured from the simulator are compared side by side with the results estimated from the equations in previous sections to verify our analysis. Finally, we use several figures to illustrate the tradeoff of the cost and gain for different approaches.

3.9.5.1 Model Validation

Table 3.3 shows the change of the total number of "DLL"s for different approaches with the group size set to be one. In the table, for the preloading-based approaches (memcpy or inline), the change of the "STD associated DLL latency" is the cost we pay for preloading, as shown in the third and fifth columns of this table. The predicted value of this part is computed using (3.37) for the memcpy approach, and (3.40) for the inline approach. The change of the total "DLL"s in the computation core (inner product and column rotation) is the gain we achieve. Without preloading, the equation for this part is (3.34); with preloading, the number of total "DLL" cycles in this part is approximately zero. Therefore, for two preloading approaches, the equation for the cycles saved in the computation core is (3.34).

The difference percentage between the measured value from the simulation trace and the predicted value from the equations is computed using the following equation:

$$Diff.Percentage = \frac{|Measurement - Prediction|}{(Measurement + Prediction)/2}. \tag{3.46}$$

From the table, we can see that the predicted value is very close to the measured value, and the difference percentage is quite small. The prediction for the "memcpy" approach has a relatively bigger difference percentage since the extra overhead of function calling is not accounted for in our simplified model.

From the naive approach to the loop unrolling approach, the only change is the inner product loop in the computation core. We expect zero change in the STD associated DLL latencies because there is no preloading. One interesting observation is the constant change of "6048" from naive approach to loop unrolling, regardless of the offchip latency (36 or 80) and the time of unrolling (4 times or 8 times). After an examination

Table 3.3: Model validation

		Latency=36		Latency=80	
		STD related DLL Latency	Computation core DLL Latency	STD related DLL Latency	Computation core DLL Latency
naive	Measured	52416	16646112	52416	39354336
memcpy	Measured	19664064	2016	42372288	2016
	Change from Naive	19611648	16644096	42319872	39354336
	Predicted change	18579456	16515072	41287680	39223296
	Diff percentage	5.41%	0.78%	2.47%	0.33%
inline	Measured	1943424	2016	4781952	2016
	Change from Naive	1891008	16644096	4729536	39354336
	Predicted change	1870848	16515072	4709376	39223296
	Diff percentage	1.08%	0.78%	0.43%	0.33%
unroll 4	Measured	46368	6711264	46368	16646112
	Change from Naive	6048	9934848	6048	22708224
	Predicted change	-	9676800	-	22450176
	Diff percentage	-	2.63%	-	1.14%
unroll 8	Measured	46368	5114592	46368	12920544
	change from Naive	6048	11531580	6048	26433792
	Predicted change	-	11273472	-	26175744
	Diff percentage	-	2.26%	-	0.98%

of the execution trace, we find it is simply due to extra instructions in the loop unrolling in the forms of one "DLL = 1" for each inner product function call. The "DLL" latency hiding phenomenon happens when a pair of "STD" and "LDD" are separated by other instructions. Therefore, we have $\binom{64}{2} \times 3 \times 1 = 6048$, given problem size 64. Since this is a post-simulation observation instead of a predication, we have put the "-" in the table. The measured change ("6048" cycles) is very small compared with the change in the computation core. So this part is of no importance to the change of the total execution cycles. Instead, we are more interested in the change of the total "DLL"s in the computation core, which is $DLL_{innerproduct} - DLL_{innerproduct\_unroll4}$ for unrolling 4 times and $DLL_{innerproduct} - DLL_{innerproduct\_unroll8}$ for unrolling 8 times and can be computed following (3.32), (3.41) and (3.42). It is noteworthy that the total "DLL"s of "loop unrolling 8 times" is not one half of the "loop unrolling 4 times" because the column rotation part is kept unchanged, although the total number of the "DLL" cycles in the inner product part is approximately reduced to one half.

3.9.5.2 Comparison of Different Approaches

Fig. 3.15 illustrates the comparison of the five approaches by decomposing total execution cycles into four parts as in (3.31). Each figure consists of five clusters of stacked bars. The first cluster shows the results of five approaches with group size one, the second one shows results for group size 2, and so on and so forth. Within each cluster, the leftmost stacked bar illustrates the four-part dissection for the naive approach, the second and third one depict the dissection for loop unrolling four times and eight times, and the fourth and the fifth represent the “memcpy” and the “inline” preloading approaches. Within each stacked bar, the brown bar (at the top), the deep blue bar (at the bottom), the light blue bar and the yellow bar (in the middle) represent the “DLL” part, the number of instructions, the “DLI” and the “DLF” part, respectively. The “DLI” part cannot be actually seen at the current scale since it is very small compared to the other parts.

We can see from the figure that the measurement matches the previous performance analysis. Firstly, all the proposed approaches except the “memcpy” approach achieve better performance – less execution cycles – than the naive approach. The “memcpy” approach performs worse than the naive approach because its scheduling of “LDD” and “STD” instructions results in L latency cycles for each “LDD/STD” pair in the data preloading section. The “inline” approach, in contrast, has a better scheduling in which every 8 “STD”s only incur $L-7$ latency cycles. Therefore, the “DLL” part of the “inline” approach is approximately one eighth of that of the “memcpy” approach.

Secondly, the way that the “DLL” part changes with the increase of the group size also confirms our analysis. For the preloading-based approaches (either “memcpy” or “inline”), the “DLL” part reduces to approximately one half when the group size doubles, which coincides with the predictions of (3.37) and (3.40). The “memcpy” approach has a large value for the “DLL” part when the group size equals one; thus its “DLL” part decreases significantly from group size one to two and becomes smaller than that of the naive approach when the group size reaches three. On the contrary, the “DLL” part of the loop unrolling-based approach does not change with the change of the group size, as indicated by (3.41) and (3.42).

Thirdly, from the naive approach to the other four approaches, the increase or decrease of the number of total instructions (the bottom bar in the stacked bar) also matches our expectation. The extra overhead for data preloading constitutes the increase of this part for the preloading-based approaches, which can be estimated according to (3.38). The “memcpy” approach has more instructions than the “inline” approach because function calls incur extra instructions. On the other hand, the loop unrolling approach reduces the total instruction cycles from the naive approach since it has a smaller number of times that the loop control statement gets executed.

Fourthly, the “DLF” part in the figures approximately does not change no matter which approaches we use. This is due to the fact that it represents the latencies related to the floating point instructions that are kept unchanged. However, the figures do show a minor increase of “DLF” from the naive approach to the preloading-based approaches. The reason is explained as follows. A code segment often includes both floating point operations and memory access operations; thus the “DLL” part and the “DLF” part sometimes overlap with each other. Without preloading, the memory accesses in the computation core have a long latency, thus the “DLF”s that overlap with “DLL”s get hidden by long “DLL”s. With preloading, the memory operations are mainly from onchip memory and the access latency becomes short; thus those “DLF”s can no longer be hidden by “DLL”s and become explicit. Nonetheless, the change of the “DLF” part is very small compared to the change of the “DLL” part and thus can be omitted safely.

Lastly, we can see that the “inline” preloading approach performs the best out of all five approaches. In fact, from the naive approach to the “inline” approach, the number of total execution cycles is reduced by 52% for latency 80, and 43% for latency 36 (with the problem size 64 by 64 and the group size 1).

Table 3.4: Total execution cycles and MFLOPS for problem size 64 by 64

	naive	“memcpy”	“unroll4”	“unroll8”	“inline”
cycles (1 thread)	60883999	70068895	38256416	33339392	29331583
MFLOPS (1 thread)	54	47	86	99	112
cycles (32 threads)	2192086	2479114	1484975	1331255	1206010
MFLOPS (32 threads)	1509	1335	2228	2485	2744

3.9.5.3 Performance on Multiple Threads

Fig. 3.16 shows the performance of different approaches on multiple threads with the problem size 64 by 64, the offchip latency 80 and the group size 1. Other parameter configurations generate similar results. Table 3.4 lists the execution cycles and the “MFLOPS” number for different approaches when the number of threads equals 1 and 32. We compute “MFLOPS” based on the histogram measurement that shows 6618532 floating point operations in one sweep. We can see that the “inline” approach performs the best and achieves 2744 MFLOPS with 32 threads.

3.10 Summary

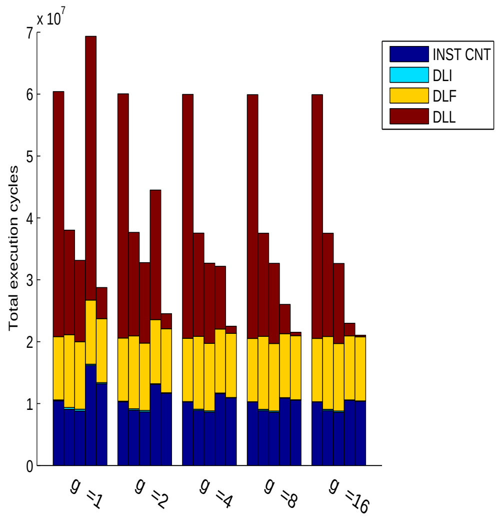
The SPACE RIP technique uses multiple receiver coils and utilizes the sensitivity profile information from a number of receiver coils in order to minimize the acquisition time. In this research, we focused on the parallel reconstruction of SPACE RIP.

Firstly, We analyzed the algorithm and identified one major loop as the program bottleneck to be parallelized. The loop level parallelization is implemented with Pthread, OpenMP and MPI and archived a near linear speedup on the Sunfire 12 CPUs SMP machine.

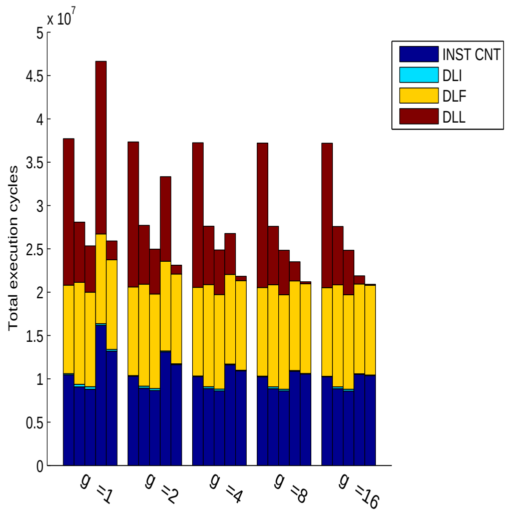
Secondly, we analyzed the one-sided Jacobi algorithm of SVD in the context of the biomedical field and proposed a rotation algorithm for complex matrices. A one-sided Jacobi algorithm for parallel complex SVD is implemented using the GaoThomas parallel ordering [78].

Thirdly, we ported the code to the new Cellular computer architecture Cyclops-64, which makes SPACE RIP one of the first biomedical applications on Cyclops-64. The speedup of the parallel SVD on Cyclops-64 achieved 43 for parallel SVD problem with matrix size 128 by 128.

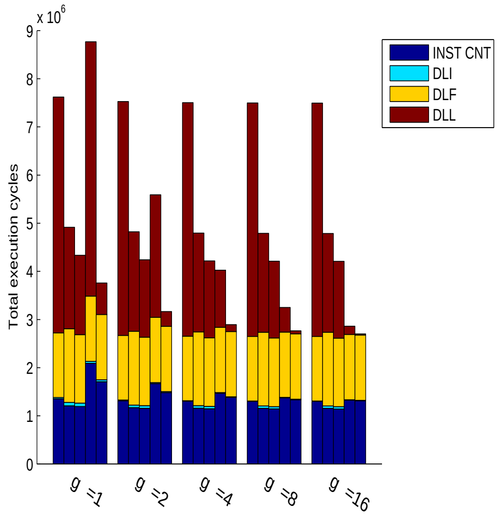
Lastly, this chapter also presented a performance model and simulation results for the preloading and loop unrolling approaches to optimize the performance of the SVD benchmark. (1), We developed a simple model and trace analyzer to dissect the total execution cycles into four parts: total instruction counts, “DLL”, “DLF” and “DLI”. This simple model allows us to study the application performance tradeoff for different algorithms or architectural design ideas. (2), We focused on the singular value decomposition algorithm and presented a clear understanding of this representative benchmark. Using a few application parameters such as matrix size, group size, and architectural parameters such as onchip and offchip latency, we developed analytical equations for different approaches such as preloading and loop unrolling. Currently, we only use offchip and onchip scratch-pad memory. The same methodology can be applied to analyze data preloading from offchip to SRAM. (3), We used a cycle accurate simulator to validate the model and compare the effects of four approaches on the “DLL” part and the total execution cycles. The simulation result and the model prediction match very well and the difference is within 5%. We find that the “inline” approach performs the best among several approaches. We also study the effect of group size on the performance and find that the total number of “DLL” cycles almost becomes one half when the group size doubles for the preloading approach.



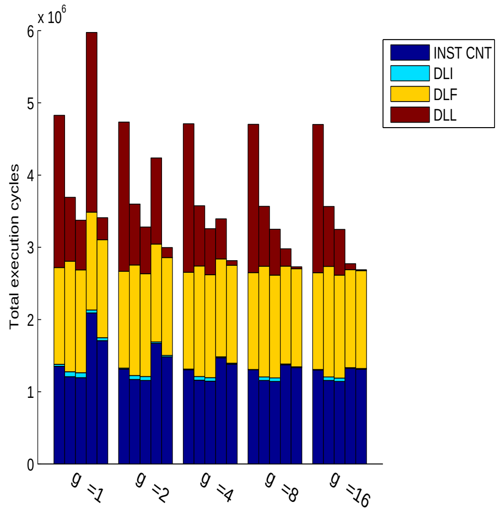
(a)



(b)



(c)



(d)

Figure 3.15: Comparison of different memory access approaches (a) Problem size 64 by 64, Latency 80, (b) Problem size 64 by 64, Latency 36, (c) Problem size 32 by 32, Latency 80, (d) Problem size 32 by 32, Latency 36

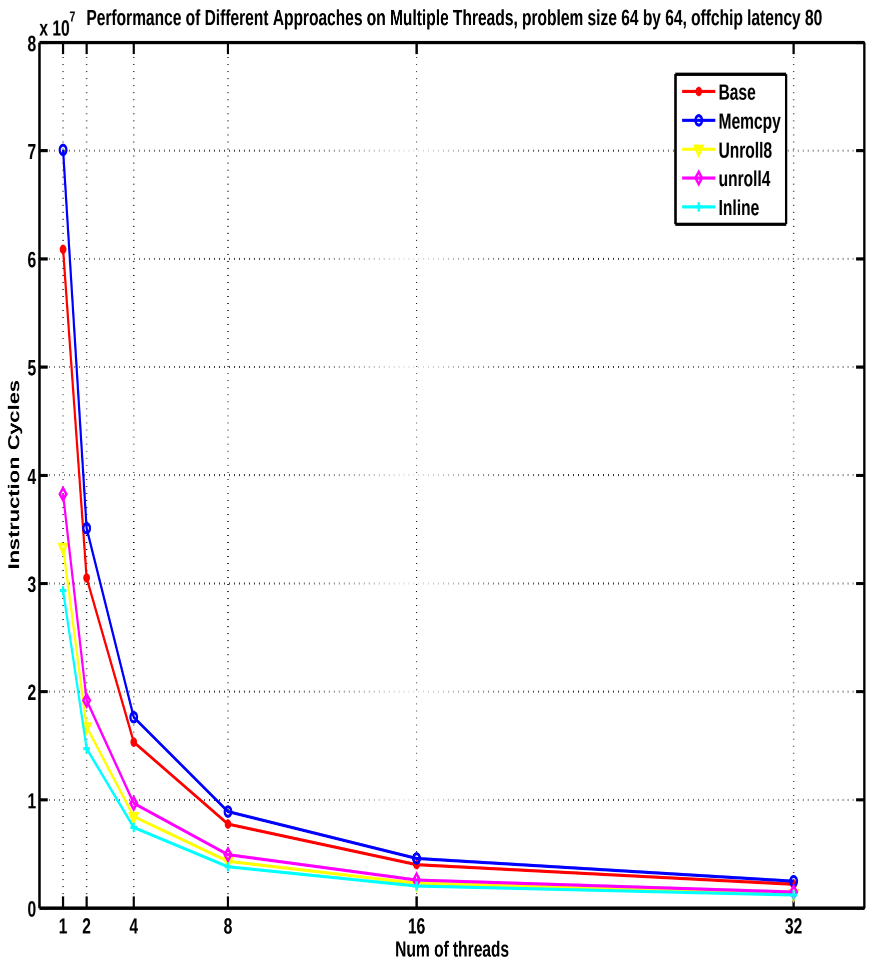


Figure 3.16: Performance of different approaches on multiple threads

Chapter 4

GENERATION OF TACTILE GRAPHICS WITH MULTILEVEL DIGITAL HALFTONING

4.1 Introduction

Humans receive all of their information about the world using one or more of the five senses [95]: the gustatory sense, the olfactory sense, the auditory sense, the tactual sense, and the visual sense. The visual sense has the highest bandwidth among the five senses, making the illustration of ideas and concepts visually through images and graphics an efficient means of communications. Loss of one of the five senses requires information to be translated from one sense to another. One possible translation is the visual to audio translation, e.g., screen reading software [96] interacts with speech synthesizers to enable visually impaired individuals to access information through voice output. Screen readers, however, can only read text, thus limiting access to graphic information.

Another possible information translation is the visual to tactile translation. The visual to tactile translation can further help with understanding graphics, including object shape and texture, while the visual to audio translation is suitable for understanding concepts. Many techniques have been proposed to translate text and graphics information into the tactile modality. In 1934, Louis Braille invented the Braille system for sightless reading and writing, regarded as one of the most significant contributions to the education of blind individuals [97]. This approach, however, is also limited to text-based information. Tactile pattern and graphics access were also made possible with the invention of the Optacon [98], which can convert a visual image into a pattern of pin vibrations that can be read with one finger. Martin *et al*. [99] developed methods to present business graphics to blind individuals, converting pie charts, bar charts and other representations into the tactile modality.

Recent work includes [100, 101], which successfully developed software algorithms for the automatic generation of tactile graphics. Meaningful information, including edges and boundaries, is identified and extracted from visual image data via a multi-step procedure. A shortcoming of simple line edge-map-based tactile graphics is that the resulting edges often intersect and do not form closed structures. This shortcoming can be addressed by segmentation procedures as well as multi-resolution edge detection [102–104]. Closed contours often yield improved representations over simple line edge-map-based representations.

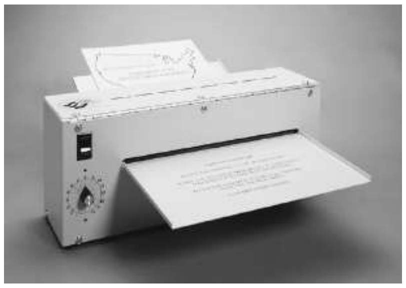
A shortcoming of binary edge maps, whether generated from an edge detection algorithm or segmentation procedures, is that features and regions are identified only by their boundaries. Such identification yields no information on the grayscale value, color, or texture of a given region. To address the issue of tactile texture representation, a tactile texture generation method for the TIE (Tactile Image Enhancer) [105] was proposed [106]. In this work, the human tactile system and tactile printing process model, along with binary halftoning algorithms, are developed and optimized for the TIE printer.

Digital halftoning is a method for rendering the illusion of continuous-tone pictures on display or printing devices that are capable of producing only a limited range of tone levels. Recent research has yielded numerous halftoning methods, many of which are reviewed in [107–109]. Halftoning algorithms generate binary on/off pixel patterns to create the visual illusion of gray level or color. This is possible due to the low pass nature of the human visual system, which averages black dots and white space to yield the perceived grayscale level. A similar approach can be taken to produce tactile textures, utilizing the low pass property of the human tactile system [106].

Although the generation of tactile patterns through halftoning procedures appears promising, numerous open problems must be addressed before the method can be used in practice. For instance, halftoning algorithms must be applied and optimized to other tactile printers, such as the TIGER embossing tactile printer [105], which is a commonly used tactile printer that has the ability to punch dots of different heights on the paper, and thus can be regarded as a multilevel system. Also, different halftoning algorithms need to be implemented to generate and compare different texture patterns, such as the deterministic textures or stochastic textures, clustered dot textures or dispersed dot textures, etc. This research addresses these specific issues.

To address the first issue, multilevel halftoning algorithms are implemented on the TIGER printer to generate tactile texture patterns. As an extension of binary halftoning, multilevel halftoning techniques are commonly adopted on printers that can generate several output levels. Therefore, multilevel halftoning is a natural choice for the TIGER printer to take advantage of the printer's variable-height punching ability. To address the second issue, four different multilevel halftoning algorithms, including AM, Bayer's, error diffusion, and green noise halftoning [108–112], are implemented, of which, AM and Bayer's are deterministic algorithms and the latter two are stochastic algorithms. Experiments are conducted to compare the halftoning-based approach with the simple commonly utilized thresholding-based approach. It is shown that the halftoning-based approaches achieved significant improvement in terms of texture pattern discrimination ability and that green noise halftoning achieved the best performance result amongst the four types of halftoning algorithms.

The remainder of this chapter is organized as follows. A brief review of tactile printing and the TIGER printer hardware and software are presented in section 4.2. In sections 4.3 and 4.4 we discuss the basics of halftoning techniques and multilevel halftoning algorithms specifically adapted for the TIGER printer. Section 4.5 presents the experiment design, protocol, and evaluation results. Finally, a brief summary is presented in section 4.7.


(a)

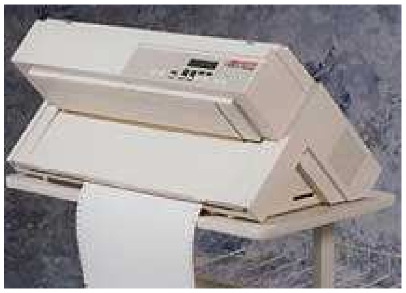

(b)

Figure 4.1: (a) TIE printer from Retro-tronics Inc., (b) TIGER printer from View Plus Technologies

4.2 Tactile Printing

The traditional way to produce tactile graphics is to collect a collage of sandpaper, cloth and other tactile materials and then glue them manually to a piece of paper. This process is time consuming and not appropriate for mass production. More recently, several types of tactile printing technology have been developed to generate tactile graphics more conveniently and effectively. In this section, currently available tactile printing devices are briefly reviewed.

4.2.1 Thermal Paper Expansion Machine

Tactile graphics can be produced by using a thermal paper expansion machine such as the SwellForm from American Thermoform [113], the Picture in a Flash (PIAF) from Pulse Data HumanWare [114], and the Tactile Image Enhancer (TIE) and TIE Junior from Repro-Tronics [105]. Fig. 4.1a shows the TIE printer from Repro-Tronics Inc.

The basic process for producing tactile images on a thermal paper expansion machine is as follows. First, prepare a very simple line graphics. It is always a good idea to extract the line or boundary information and print only the main features of computer graphics [103]. Next, use a normal ink or laser printer to print the graphics directly on the front of the thermal expansion paper. A photocopier can also be used to transfer an

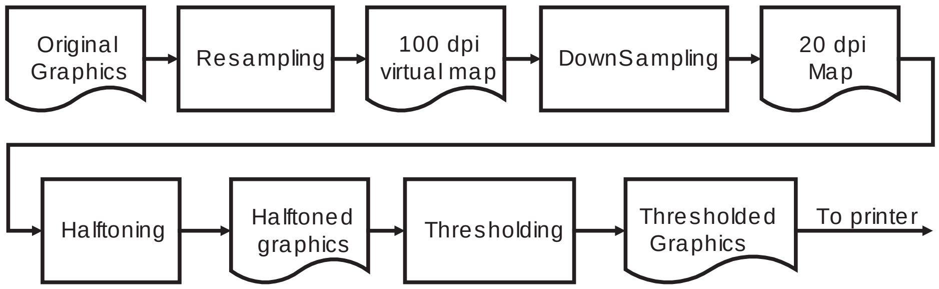


Figure 4.2: Graphics printing pipeline

image from normal paper to thermal expansion paper. Finally, feed the thermal expansion paper through the TIE printer. The micro-capsule coating on the thermal expansion paper is heat reactive. The black lines or images printed on the thermal expansion paper absorb more heat than the surrounding areas when exposed to a heat source, causing the underlying capsules to grow, yielding the raised lines, areas, and symbols on the thermal expansion paper.

The thermal imaging pen from Repro-Tronics [105] can also be used to produce tactile graphics. This pen is a tool that allows a person to draw a raised image directly on the thermal expansion paper by hand. The tip of the pen is heated and causes the paper to swell at the point of contact, producing a raised image.

4.2.2 TIGER Embossing Printer

Unlike thermal paper expansion machines, the TIGER printer directly punches dots of variable heights on paper or plastic media. It is a Windows-based printer developed by View Plus Technologies [115], Fig. 4.1b. The TIGER printer prints text in any computer Braille font or the new DotsPlus font. Graphical content is printed as well, whereas complex graphics must be hand-edited to produce a comprehensible tactile form. The TIGER printer is the main experimental platform utilized in this research. In the remainder of this section, the internal mechanisms of the printer's graphics printing pipeline are discussed.

The graphics printing pipeline is show in Fig. 4.2. In this figure we illustrate where we insert the digital halftoning step in the proposed scheme. In the graphics printing pipeline, the graphics to be printed are first converted to a 100 dpi virtual map. At this resolution, various image processing algorithms, such as image segmentation algorithms, are applied to generate an appropriate representation [116].

After the resampling step, the downsampling step converts the 100 dpi virtual map to a 20 dpi virtual map. The downsampling step is necessary for tactile image generation. Research [100, 117] shows that the minimum tactually discernible grating resolution for a human fingertip is only 1.0 mm, indicating that the resolution of a tactile image should be somewhat finer than 1 dot/mm. This corresponds to a resolution of approximately 25.4 dpi. For comparison, images for visual displays have resolutions of 72 dpi (CRT) to 2400 dpi (laser printer). Therefore, to convert these visual images into the tactile form, the high resolution images must be downsampled to approximately 20 dpi. In the current printer implementation, there are two methods for downsampling an image, either a mean or max filter. The user can choose one of these two options via the TIGER driver control program. Both filters work on a 5 by 5 block, where the filter windows are tiled upon the 100 dpi virtual map. The mathematical representation of the mean filter is given by (4.1) and the max filter given by (4.2):

$$y(n) = \frac{1}{S} \sum_{p \in W} x(p), \tag{4.1}$$

$$y(n) = \max_{p \in W} x(p), \tag{4.2}$$

where W denotes the neighborhood window of the pixel $y(n)$, $x(p)$ denotes the gray value of pixel p in the high resolution image, and $y(n)$ denotes the gray value of pixel n in the low resolution image. The window size S of W is 5×5.

The proposed multilevel halftoning algorithm is inserted between the downsampling step and the thresholding step, which means it manipulates the pixels on the 20 dpi bitmap. The thresholding step quantizes the input gray levels in the 20 dpi virtual map to gray levels that can be produced by the printer. The TIGER printer can punch dots

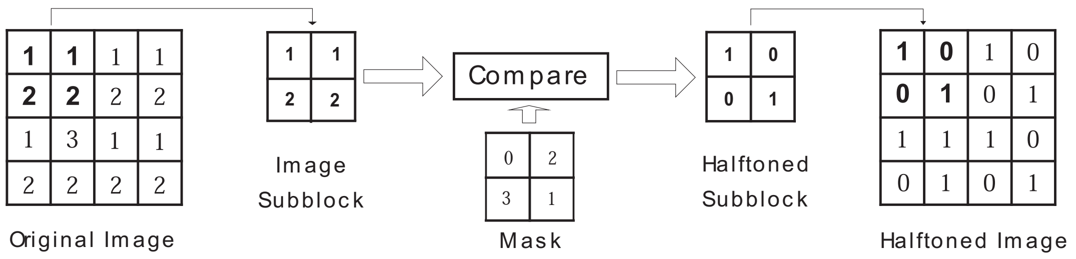


Figure 4.3: Example of mask-based halftoning

with 8 different heights. Thus the simple thresholding-based printer driver quantizes the gray level [0, 1] to 8 different output levels. The detailed specifications of the dot height are beyond the scope of this dissertation. Note that if halftoning is applied on the 100 dpi virtual map, then the output of the halftoning step is further processed at the down-sampling step, and the halftoned pattern is destroyed. Therefore, we apply the halftoning algorithms on the 20 dpi virtual map instead of the 100 dpi virtual map, as reflected in Fig. 4.2. In our scheme, the halftoning processing step outputs bitmaps with different gray levels, which are equal to the output levels of the quantizer, and are passed through the thresholding step and sent directly to the printer.

4.3 Binary Halftoning Algorithms

Many effective algorithms have been proposed in the field of halftoning. In this section, several popular halftoning techniques are briefly reviewed, including AM halftoning, Bayer's halftoning algorithm, error diffusion halftoning and green noise halftoning.

4.3.1 AM Halftoning

"Amplitude modulated" (AM) halftoning [108] is a clustered dot ordered dithering algorithm. Gray level is presented by varying the size of the dots that are printed along a regular lattice. This method is primarily used for printers that have difficulty producing isolated single pixels. AM halftoning is a mask-based halftoning technique, Fig. 4.3. The mask-based algorithm works as follows: a continuous-tone original image is compared to the mask, where each pixel of the original image is converted directly into a binary dot based on a pixel-by-pixel comparison with the mask. Pixels with a gray level greater than their corresponding threshold in the mask are converted to “on” pixels in the final halftoned image, while pixels less than the corresponding thresholds are turned “off”. The mask is tiled end-to-end on the original image. The mask is generated such that the output “on” pixels tend to cluster together. One example of the AM mask [109] is shown in Fig. 4.4a. The AM halftoned pattern of a gray level ramp is shown in Fig. 4.5b.

4.3.2 Bayer’s Halftoning Algorithm

Bayer’s algorithm [110] is a dispersed dot ordered dithering technique. This technique is also a mask-based method, but it turns pixels “on” individually without grouping them into clusters. In Bayer’s mask, consecutive thresholds are dispersed as much as possible, an example [108] of which is shown in Fig. 4.4b. By maintaining the size of printed dots for all gray levels as an individual pixel, dispersed dot halftoning techniques vary the spacing between printed dots according to the gray level, earning the name “frequency modulated” or FM halftoning. The Bayer’s method produces periodic structures. In the visual case, these structures introduce an unnatural visual appearance. However, the periodic structure may be helpful for perception in the tactile case, which is why we include Bayer’s algorithm here and implement it into the TIGER printer driver. Like AM halftoning, the Bayer halftoning algorithm is a deterministic halftoning algorithm. The Bayer’s halftoned pattern of a gray level ramp is shown in Fig. 4.5c.

4.3.3 Error Diffusion Halftoning

The error diffusion halftoning technique was proposed by Floyd and Steinberg [111]. This algorithm is a stochastic dispersed dot halftoning. Error diffusion also keeps the dot size fixed as a single pixel, and the required illusion of continuous tone is achieved by varying the distance between printed dots. This process adopts a stochastic method for quantizing the gray level of each pixel. The quantization error produced in each of

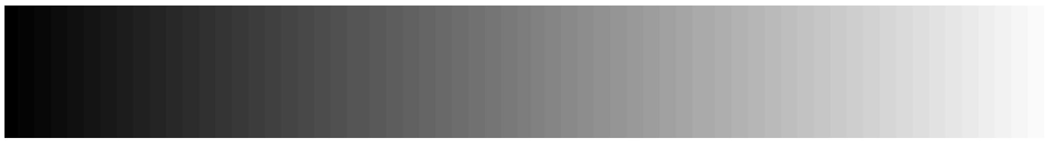
0	8	22	26	30	19	5	1
7	14	37	46	47	38	13	6
21	36	51	52	53	48	39	20
29	45	59	60	61	54	40	27
25	44	58	63	62	55	41	31
16	35	50	57	56	49	32	17
10	15	34	43	42	33	12	11
2	9	23	28	24	18	4	3

(a)

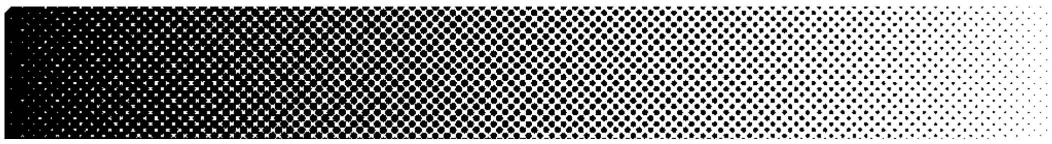
0	58	14	54	3	57	13	53
32	16	46	30	35	19	45	29
8	48	4	62	11	51	7	61
40	24	36	20	43	27	39	23
2	56	12	52	1	59	15	55
34	18	44	28	33	17	47	31
10	50	6	60	9	49	5	63
42	26	38	22	41	25	37	21

(b)

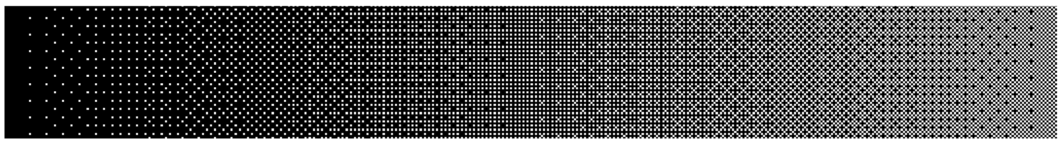
Figure 4.4: Halftoning mask (a) AM (b) Bayer's



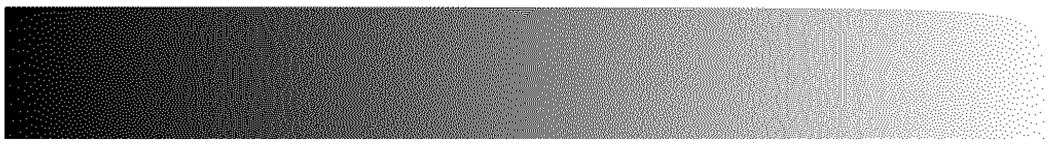
(a)



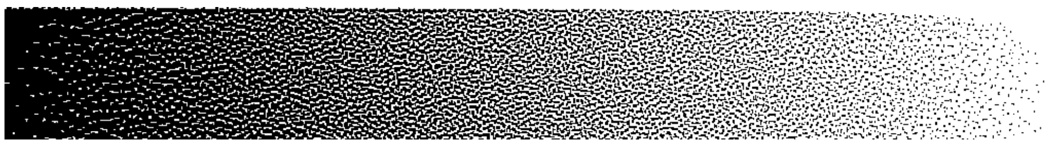
(b)



(c)



(d)



(e)

Figure 4.5: Binary halftoned ramp (a) original, (b) AM halftoning, (c) Bayer's halftoning, (d) Error diffusion (Blue Noise) halftoning, (e) Green Noise Halftoning

the single pixel operations is distributed amongst the neighborhood of the pixel to be processed. This algorithm is illustrated in the block diagram shown in Fig. 4.6a. The error diffusion halftoned pattern of a gray level ramp is shown is Fig. 4.5d.

The pattern generated by error diffusion shows an irregularly dispersed dot pixel pattern and is referred to as the blue-noise halftone [107]. This name results from the fact that the halftoned pattern is composed exclusively of high frequency spectral components. The error diffusion algorithm devised by Floyd and Steinberg, shown in Fig. 4.6a, can be mathematically represented as follows [108]:

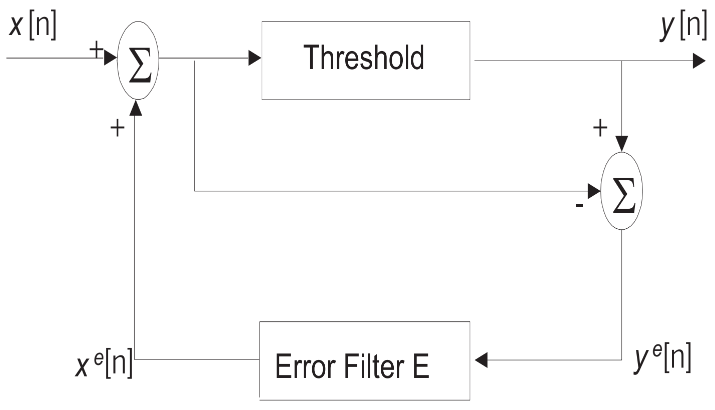
$$y[n] = \begin{cases} 1 & \text{if } (x[n] + \overrightarrow{e}^T \overrightarrow{y^e}[n]) \geq 0 \\ 0 & \text{else} \end{cases}, \tag{4.3}$$

where coefficient vector $\overrightarrow{e}$ of error filter E is $\overrightarrow{e} = [e_1, e_2, ..., e_N]^T$, and the output is written as $\overrightarrow{y}[n] = [y[n-1], y[n-2], ..., y[n-N]]^T$, $\overrightarrow{y^e}[n] = [y^e[n-1], y^e[n-2], ..., y^e[n-N]]^T$, which follows from $y^e[n] = y[n] - (x[n] + x^e[n])$ and $x^e[n] = \overrightarrow{e}^T \overrightarrow{y^e}[n]$. The input pixel under consideration in this expression is $x[n]$.

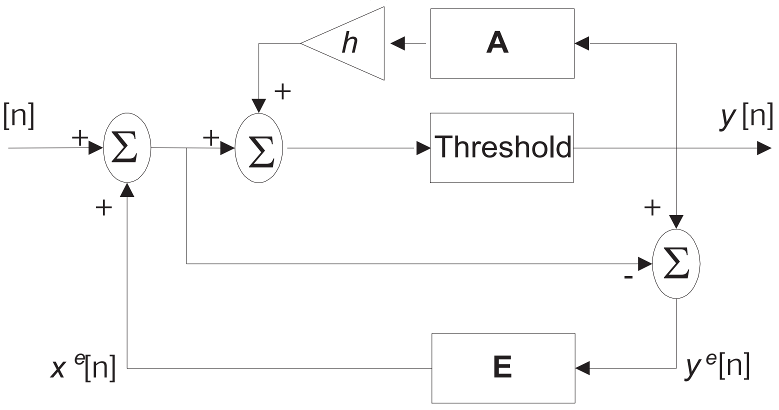
An ideal printer is able to output patterns that are composed of perfect square black dots. In high quality printing situations, where this effect is true to a certain extent, blue noise halftoning is considered the optimum technique for minimizing visibility [118] and maximizing the apparent spatial resolution [108, 119]. The error diffusion technique has a certain characteristic that makes it superior to AM methods: it can present fine detail with high spatial resolution.

4.3.4 Green Noise Halftoning

Green noise halftoning [112] is a combination of the clustered and dispersed dot halftoning techniques and is also called an AM-FM hybrid method. The green noise model describes the spatial and spectral characteristics of visually pleasing dither patterns composed of a random arrangement of clusters that vary with gray level in both their size and shape, as well as distance between the clustered dots. The term “green” refers to the



(a)



(b)

Figure 4.6: Schematic representation of halftoning algorithms (a) Error diffusion, (b) Green noise halftoning

mid-frequency content of the corresponding halftone patterns, as green light is the mid-frequency of white light. The green noise halftoned pattern of a gray level ramp is shown in Fig. 4.5e.

In green noise halfoned images, the minority pixel clusters are distributed homogenously. The green noise halftone is generated as an extension to the error diffusion technique proposed by Levien [112] and referred to as the error diffusion with output dependent feedback. The block diagram of the algorithm is shown in Fig. 4.6b. In this algorithm, a weighted sum of previous output pixels is used to vary the threshold. This makes minority pixels more likely to occur in clusters. The amount of clustering is controlled through the hysteresis constant h. Large values of h cause large clustering and small values lead to smaller clusters.

Levien's algorithm is precisely defined as follows [108]:

$$y[n] = \begin{cases} 1 & \text{if } (x[n] + \overrightarrow{e}^T \overrightarrow{y^e}[n] + h\overrightarrow{a}^T \overrightarrow{y}[n]) \geq 0 \\ 0 & \text{else} \end{cases}, \tag{4.4}$$

where $\overrightarrow{a} = [a_1, a_2, ..., a_N]^T$, $\overrightarrow{e} = [e_1, e_2, ..., e_N]^T$, $\sum_{i=0}^{N} a_i = 1$, $\sum_{i=0}^{N} e_i = 1$, and the output $\overrightarrow{y}[n]$ and $\overrightarrow{y^e}[n]$ are defined as before. The coefficient vector $\overrightarrow{e}$ of error filter E, coefficient vector $\overrightarrow{a}$ of hysteresis filter A, and hysteresis constant h can take on a wide range of values, including special cases such as Floyd-Steinberg [111], Jarvis [120] and Stucki [121] filter coefficients.

4.4 Multilevel Halftoning Algorithms

In ink and laser printing, technologies that can generate more than two output levels are becoming increasingly common [109]. The image quality studies by Huang *et al.* [122] demonstrated that a few intermediate output levels can provide a substantial improvement to halftoned images. Therefore, multilevel halftoning [123, 124] extensions from traditional binary halftoning algorithms are also becoming more common. In this section, research on multilevel halftoning is briefly reviewed. Two simple extensions are adopted to extend the binary halftoning algorithms to multilevel tactile halftoning.

4.4.1 Mask-based Multilevel Halftoning

AM halftoning and Bayer's technique use mask screening on a regular lattice. The same approach is used to extend these halftoning algorithms from binary to multilevel. Simply put, we divide the normalized gray level [0, 1] into N small intervals with uniform length. In the TIGER printer case, N can be 2 through 8. Each small interval is mapped to a [0, 1] range, and traditional binary halftoning is applied upon this interval. This approach is illustrated in Table 4.1. It is noteworthy that $mask[r][s]$ in the table is normalized into the range of [0, 1]. Fig. 4.7 shows the simulation result and the histogram of the AM multilevel halftoning algorithm. Similar results for Bayer's and other halftoning algorithms are not shown due to space limitation. It is apparent from the figure that for input gray levels between output level g_i and g_{i+1}, the halftoned pattern is the clustered dot combination of only these two output levels. The halftoned ramp patterns generated with multilevel AM and Bayer's algorithms are shown in Fig. 4.8b and Fig. 4.8c respectively.

4.4.2 Error-Diffusion-Based Multilevel Halftoning

The error diffusion and green noise halftoning techniques adopt similar algorithm structures, as illustrated in Fig. 4.6. Thus, the same multilevel halftoning extension is adopted for these algorithms. In the multilevel halftoning extensions, the binary thresholding [125] in the block diagram is replaced by a multilevel quantizer, as shown in Fig. 4.9. Note that only the extension to the error diffusion algorithm is shown. The extension to the green noise halftoning algorithm is similarly straightforward and thus not shown. The possible output of the quantizer is one of the 8 levels that the printer is able to print. As shown in the graphics printing pipeline, our algorithm is applied on the 20 dpi virtual

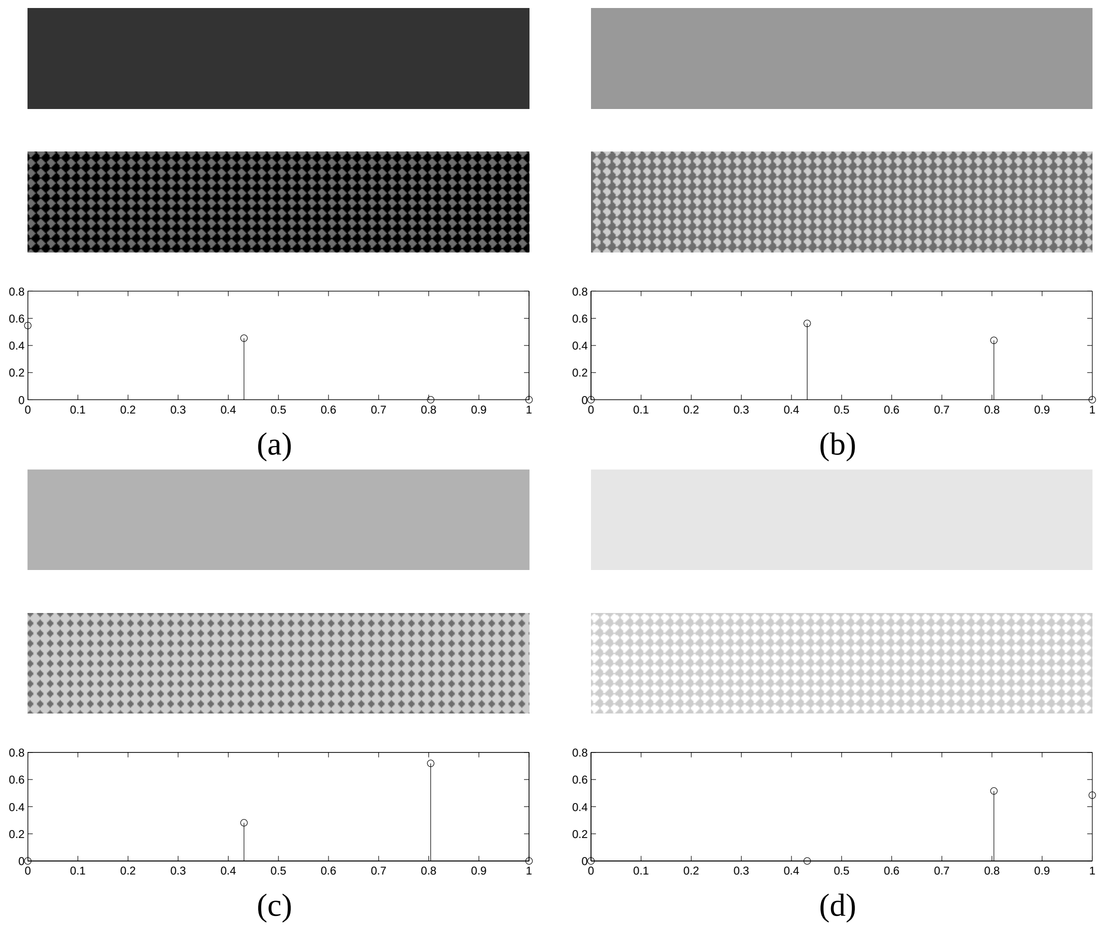


Figure 4.7: Four-level AM halftoning for solid grayscale pattern with grayscale level (a) 0.2 (b) 0.6 (c) 0.7 (d) 0.9; for each gray level, the top row is the input image, the middle row is the halftoned pattern (amplified to show the detail) and the bottom row is the histogram of the halftoned pattern

Table 4.1: Algorithm pseudocode for mask-based multilevel halftoning

```
For all the pixels $input\_image[m][n]$, do the following:
{
    for $(i = 1; i < N; i++)$
    if $((input\_image[m][n] \geq g_i)$ AND $(input\_image[m][n] < g_{i+1}))$
    {
        $low = g_i;$
        $high = g_{i+1};$
        $break;$
    }
    $r = (m \% mask\_row);$
    $s = (n \% mask\_col);$

    if      $(((input\_image[m][n] - low) >= mask[r][s] * (high - low))$
        $output\_image[m][n] = high;$
    else
        $output\_image[m][n] = low;$
}
```

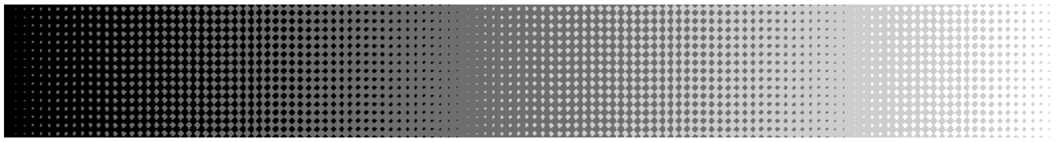
map, and the output of the halftoning is sent to the printer without any further processing. Mathematically, the quantizer is expressed by the following equations [109]:

$$x_a[n] = x[n] + x^e[n], \tag{4.5}$$

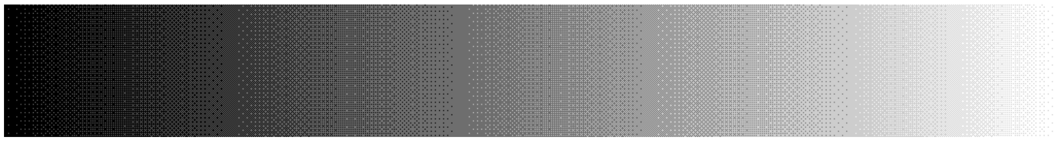
$$y[n] = \begin{cases} g_1 & \text{if } x_a[n] < T_1 \\ g_2 & \text{if } T_1 \leq x_a[n] < T_2 \\ g_3 & \text{if } T_2 \leq x_a[n] < T_3 \\ \vdots & \vdots \\ g_{N-1} & \text{if } T_{N-2} \leq x_a[n] < T_{N-1} \\ g_N & \text{if } T_{N-1} \leq x_a[n] \end{cases}, \tag{4.6}$$

where the tone level g_i above is the i-th output level of the TIGER printer, T_i is the i-th threshold value, and N is the number of output levels. The halftoned ramp patterns generated with multilevel error diffusion and green noise halftoning algorithms are shown in Fig. 4.8d and Fig. 4.8e.

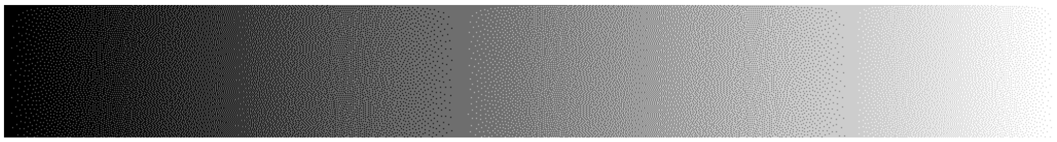
(a)



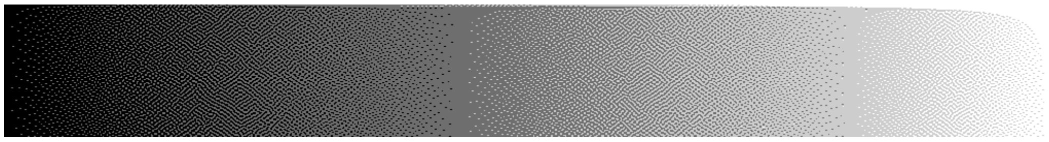
(b)



(c)



(d)



(e)

Figure 4.8: Multilevel halftoned Ramp with four output levels (a) original , (b) AM halftoning , (c) Bayer's Halftoning, (d) error diffusion (bue noise) halftoning, (e) green Noise Halftoning

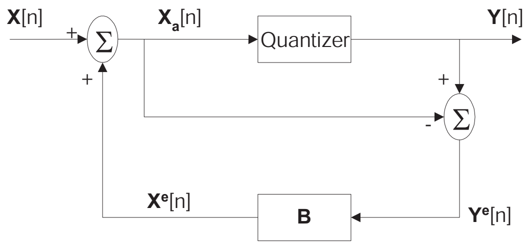


Figure 4.9: Multilevel error diffusion

4.5 Implementation

Unlike the halftoning pattern generation for the TIE printer [105], the halftoning algorithms are directly inserted into the TIGER printer driver, thus enabling this printer to work with every Windows application program, such as word processing software, graphics software and Internet browsers. The architecture of the general Windows graphics printing driver is shown in Fig. 4.10.

Generally speaking, the graphics printer driver is a software interface between the graphics rendering engine and the printing device. The input to the printer driver is sent from the Windows applications through the graphics engine. Microsoft provides a sample driver called “UniDRV”, which consists of a working driver that can be adjusted to the specific requirements of the corresponding printer. A printer driver programmer can customize the UniDRV by providing a user mode dynamic-link library (DLL) in which the customized versions of some graphics rendering functions are implemented. This user DLL is referred to as a “rendering plug-in”. In our case, four different halftoning algorithms are implemented in the user mode DLL, which can be called directly by the UniDRV driver. This procedure is called “hook out a Windows function”. More information on device driver programming can be found in [126].

The parameters for different algorithms are selected experimentally. In the current experiment setup, AM halftoning and Bayer’s halftoning adopt 8×8 masks, as shown in Fig. 4.4. The error filter weight matrix for error diffusion and green noise halftoning adopt the weight parameters [111] of the classical Floyd’s and Steinberg’s algorithm, as

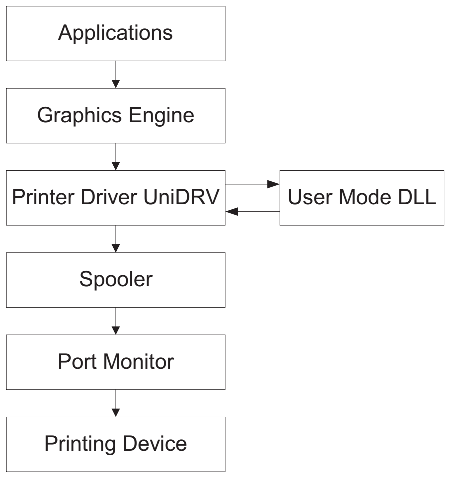


Figure 4.10: Windows graphics printing architecture

$$\begin{array}{ccc} & \bullet & 7/16 \\ 3/16 & 5/16 & 1/16 \end{array}$$

Figure 4.11: Floyd's and Steinberg's error filter

shown in Fig. 4.11. The hysteresis for green noise halftoning h is set at 0.8.

4.6 Evaluation

Experiments were conducted to evaluate and compare different algorithms, including the original thresholding-based approach and various halftoning algorithms. Since the main aim of this research is to use various halftoning algorithms to generate texture patterns that can represent different gray levels, experiments were designed to focus on the ability of different algorithms to represent and discriminate different gray levels. In this section, the test algorithms, test material generation, experimental procedure, and experimental results are presented.

4.6.1 Test Algorithms

There are four types of halftoning algorithms to be evaluated: AM, Bayer's, Error diffusion and Green noise halftoning algorithms. For each algorithm, we can use binary halftoning, in which the output pattern is composed of either "no dot" (blank paper) or highest dot. Alternatively, we can use three-level halftoning, in which the output pattern is composed of three possibilities: "no dot", dot with medium height, or highest dot. Similarly, four-level or five-level halftoning algorithms can also be implemented.

Of interest is the determination of whether or not the multilevel halftoning algorithms generate better tactile patterns in the sense of better discrimination ability and effectiveness. Also of interest is the optimal number of output levels, i.e., which of the binary, three-level or four-level halftoning, etc is the best choice. Our hypothesis is that more output levels do not necessarily result in better discrimination ability. If we use too many output levels, then the height difference between two neighboring output levels becomes negligible. Since the multilevel halftoning algorithms are designed such that a gray level is represented by the combination of two neighboring output levels, the texture may not be apparent enough to be discriminated with the sense of touch if the two neighboring output heights are insufficiently distinct.

Therefore, different algorithms with different output levels are included in the experiments in order to answer the previous questions and to test whether our hypothesis is correct. Preliminary tests show that tactile patterns generated with five output levels are insufficiently distinct to be discriminated. Therefore, five-level algorithms are not included in the presented result. The algorithms tested are listed in the first column of Table 4.3.

4.6.2 Test Material

The experiments conducted are discrimination experiments in which subjects explore freely left and right tactile image pairs and tell whether they are different or not. Therefore, the test materials include image pairs. Each image is a square of one specific

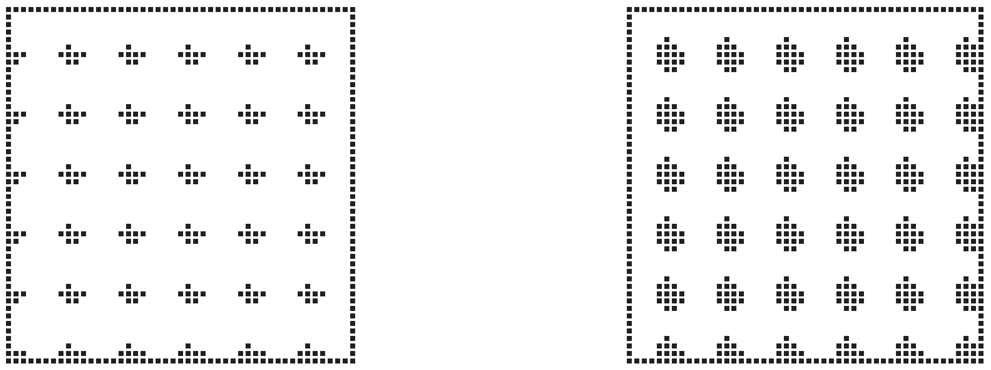


Figure 4.12: Left and right texture pattern of AM halftoning

gray level. Some of the pairs are the same (with the same gray level), and some of the pairs are different. One example is shown in Fig. 4.12, in which the left and right patterns are generated using AM halftoning.

The image pairs are generated as follows. Step 1: the gray level range from 254 to 0 is quantized into 7 different ranges. The middle points of the 7 different ranges are selected and denoted as $I1, I2, ..., I7$. The gray level 255 is denoted as $I0$. For the original thresholding method, this input data set $I0$ to $I7$ generates solid square patterns with dot heights from level 0 to level 7, where level 0 is blank paper and level 7 is the highest dot. For the halftoning algorithm, the gray levels $I0$ through $I7$ are represented by different halftoning texture patterns. Step 2: square patterns with gray levels $I1$ through $I7$ are printed using each of the 13 different algorithms. I0 is not included since it is represented by plain paper in all algorithms. The pair combinations are listed in Table 4.2. The total number of pairs for each algorithm is 13.

4.6.3 Experimental Procedure

The experimental design focuses on discrimination ability. Discrimination is an important perceptual task extensively studied in the field of psychophysics. It addresses

Table 4.2: Test pairs

Type of Pair	Combinations
Left and right Different	(I1, I2), (I2, I3), (I3, I4), (I4, I5), (I5, I6), (I6, I7)
Left and right Same	(I1, I1), (I2, I2), (I3, I3), (I4, I4), (I5, I5), (I6, I6), (I7, I7)

the question “Is one stimulus different from another one?”. In our experiments, the test subjects explore two tactile objects, only by the sense of touch, and indicate whether they think they are the same or not.

Ten sighted test subjects participated voluntarily in the experiment. Seven subjects are male and three subjects are female. It is widely believed that touch sensitivity varies little from subject to subject, and that there is no statistical difference between the sighted and unsighted populations [127, 128]. Therefore, information on how individuals with visual impairment perceive can be inferred from the sighted subject results.

Each subject was asked to perform a discrimination task using one complete set of 13 pairs per algorithm $\times$ 13 algorithms. Subjects were seated at a table, blindfolded and presented with a set of 13×13 sheets in random order. Subjects were briefly introduced to the basic features of different algorithms at the beginning of the experiments. For each sheet, subjects freely explored the pairs of tactile images on the sheet for a time period. This gives the subjects enough time to glean information about the texture/gray level of the images. Then the subjects were asked to report whether the images felt the same or different. Subjects also could make a guess if they could not say one way or the other. During this procedure, the responses were recorded.

4.6.4 Experimental Results, Observations and Analyses

The experimental results are summarized in Table 4.3 and depicted in Fig. 4.13. For each of the 13 algorithms, 13 images pairs $\times$ 10 subjects constitute the total number of experiments. Out of the 130 responses, only the number of correct answers are counted, and the percentage of correct answers is listed in the table. Analysis of variance is denoted by p, and used to compare the different halftoning algorithms with the original thresholding-based approach and with chance (50%).

There are several observations from the table that can be noted. For instance, it is noteworthy that a correct response of 50% is expected for a pure guess, and 100% is expected for a perfect performance. From Table 4.3, it can be seen that the original thresholding algorithm has approximately 50% correct responses. This is due to the fact that no texture is generated by the thresholding approach, and it is difficult, if not impossible, to discriminate between different gray levels.

In addition, the correct response percentage for halftoning-based approaches, especially the binary and three-level halftoning algorithms, is higher than the original algorithm with statistical significance, as reflected by the p values. It is not concluded whether three-level or binary algorithms are better. For Bayer's and green noise halftoning, three-level algorithms are slightly better than binary algorithms, while for AM and error diffusion, binary halftoning is slightly better.

Moreover, it can be seen from the table that the percentage values of the four-level algorithms are close to 50%. Also, the preliminary experiments indicate no significant difference between five-level algorithms and chance (results not shown in the table). As stated before, the reason is due to the reduced difference between two neighboring output levels.

Also, the comparison between different output levels within the same algorithm is illustrated in Table 4.4. It can be seen that for AM halftoning, the binary algorithm is better than the three-level and four-level algorithms with statistical significance. However, for the other three halftoning algorithms, it cannot be established that there is significant difference among binary, three-level and four-level algorithms.

Lastly, it can also be seen that AM and green noise halftoning are slightly better than the Bayer's and error diffusion algorithms. This is probably due to the fact the both AM and green noise can generate clustered dots that are easily discernable by the sense of touch. It is noteworthy that the three-level green noise halftoning algorithm has an correct response greater than 80%, which is a significant improvement from the simple thresholding-based method. This may be due to the fact that the three-level green noise algorithm generates a more prominent texture in certain local gray level ranges since it changes both the cluster size and cluster distribution to represent different gray levels. This result is in agreement with [106], which reported green noise halftoning as the best halftoning algorithm for TIE generated output. Results presented there and here suggest that the green noise algorithm parameters may be effectively optimized for tactile halftoning. Such optimization is the focus of future work.

Table 4.3: Comparison of different halftoning algorithms

Algorithm	Percentage of Correct Response	p (vs. Original)	p (vs. Chance)
Original	49.23%	1.00e+000	5.5e-001
Binary AM	75.38%	2.19e-007	1.0e-007
Three Level AM	72.31%	6.91e-006	4.9e-006
Four Level AM	63.08%	1.04e-005	2.2e-006
Binary Bayer's	64.62%	5.06e-006	1.2e-006
Three Level Bayer's	70.77%	4.46e-007	1.5e-007
Four Level Bayer's	56.92%	4.03e-004	3.1e-005
Binary Error Diffusion	67.69%	2.24e-007	3.1e-008
Three Level Error Diffusion	64.62%	5.06e-006	1.2e-006
Four Level Error Diffusion	58.46%	6.08e-005	2.6e-006
Binary Green Noise	75.38%	3.48e-006	2.6e-006
Three Level Green Noise	81.54%	6.84e-008	3.9e-008
Four Level Green Noise	63.08%	1.04e-005	2.2e-006

4.7 Summary

In this work, the major contributions are as follows: (1), We introduced digital halftoning algorithms into the TIGER printer to generate tactile graphics. Four different

Table 4.4: Comparison of different output levels within same algorithm

Algorithm	Level Comparison	*p* value	Algorithm	Level Comparison	*p* value
AM	Two vs. Three	3.5e-006	Bayer's	Two vs. Three	5.1e-001
	Two vs. Four	6.8e-008		Two vs. Four	2.8e-003
	Three vs. Four	2.5e-001		Three vs. Four	3.2e-002
Error diffusion	Two vs. Three	7.4e-002	Green Noise	Two vs. Three	2.9e-001
	Two vs. Four	5.0e-003		Two vs. Four	7.9e-004
	Three vs. Four	1.1e-004		Three vs. Four	2.0e-002

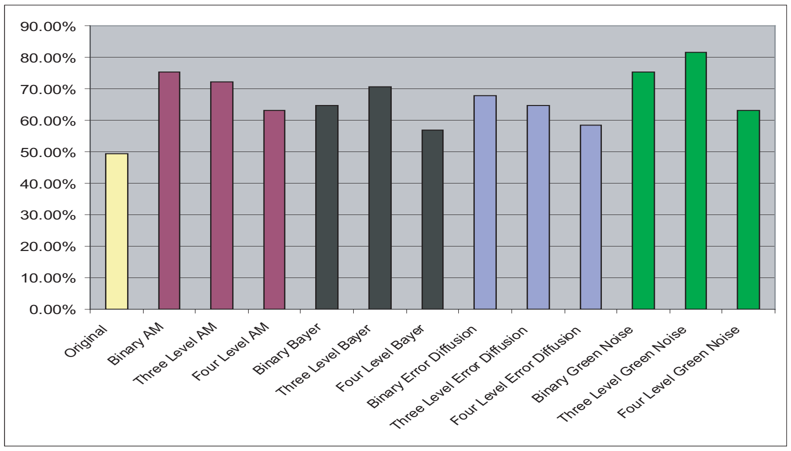


Figure 4.13: Comparison of different halftoning algorithms

halftoning algorithms are implemented into the TIGER printer driver. (2), According to the specifics of the TIGER printer, traditional binary halftoning algorithms are extended to multilevel algorithms. (3), Experiments are conducted to show that the new approach can generate better tactile graphics; tentative conclusions about which algorithms are more suitable for the TIGER printer are drawn.

Chapter 5

CONCLUSIONS AND FUTURE DIRECTIONS

In this dissertation, we concentrated on performance optimization of three representative applications from the bioinformatics or biomedical area using state-of-the-art computer architectures and technologies. We believe the methodologies adopted in the study of these three applications can be applied to other interesting applications as well.

First of all, we proposed a new task decomposition scheme to reduce data communication and generated a scalable and robust cluster-based parallel Hmmpfam using the EARTH (Efficient Architecture for Running Threads) model. The methodology is to balance the computation and communication in cluster-based computing environments.

Secondly, we used the real biomedical application SPACE RIP as a context and focused on the core algorithm SVD. We implemented the one-sided Jacobi parallel SVD on Cyclops-64 to exploit the thread-level parallelism. We also developed a performance model for the dissection of total execution cycles into four parts and used this model to compare different memory access approaches. We observed a significant performance gain with the combination of these parallelization and optimization approaches.

Our work on the parallelization and optimization of SPACE RIP is one of the attempts to adapt to the era of multicore processor designs. The new trend of multicore processors forces a fundamental change of software programming models. Many applications have enjoyed free and regular performance gains for several decades, sometimes even without releasing new software versions and doing anything special, because the CPU manufactures have enabled ever-faster mainstream systems. With the multicore processors, the “free lunch” is over [9]. Multithreaded software must be developed to fully exploit the power of multicore processors. Moreover, efficiency and performance tuning will get more important. With the results and conclusions from our optimization of the SPACE RIP application, future extensions and optimizations to existing programming languages and compilers may be developed.

Finally, we adapted different halftoning algorithms to a specific tactile printer and conducted experiments to compare and evaluate them. The idea is to find a good way to utilize modern computer technologies and image processing algorithms to convert graphics to multilevel halftoning texture patterns that are manually perceivable by individuals with visual impairment. We concluded that the halftoning-based approach achieves significant improvement in terms of its texture pattern discrimination ability and that the green noise halftoning performs the best among different halftoning algorithms.

This dissertation shows the promise of using parallel computing technology and digital imaging algorithms to find better solutions for real applications. At the conclusion of our research, we found that following areas have opened up for further exploration. First of all, in the direction of combining bioinformatics/biomedical applications and parallel computing, we may focus on other interesting and challenging applications. Porting applications, such as multiple sequence alignment (MSA), to Cyclops-64 may generate interesting findings, such as novel parallel schemes and insights for architecture designs. Secondly, the current halftoning-based approach for tactile graphics can be further extended to process color images using digital color halftoning techniques. Digital halftoning can also be integrated with other tactile imaging techniques, such as image segmentation and edge detection, to generate the texture in the segmented regions.

In conclusion, we feel that the combination of parallel computing and bioinformatics/biomedical algorithm/applications is indeed an interesting multi-disciplinary area. It is worthy to take long term efforts to develop innovative approaches and provide better solutions to existing and emerging problems.

BIBLIOGRAPHY

[1] T. Sterling, D. Becker, and D. Savarese, "BEOWULF: A parallel workstation for scientific computation," *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pp. 11–14, 1995.

[2] The 24th TOP500 Supercomputer List for Nov 2004. [Online]. Available: http://www.top500.org

[3] K. B. Theobald, "EARTH: An efficient architecture for running threads," Ph.D. dissertation, McGill University, Montreal, 1999.

[4] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, and G. R. G. et al., "A design study of the EARTH multiprocessor," *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 59–68, 1995.

[5] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, no. 1, pp. 29–63, 2003.

[6] G. R. Gao, L. Bic, and J.-L. Gaudiot, Eds., *Advanced Topics in Dataflow Computing and Multithreading*. IEEE Comp. Soc. Press, 1995, book contains papers presented at the Second Intl. Work. on Dataflow Computers, Hamilton Island, Australia, May 1992.

[7] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, 1997.

[8] G. A. Alverson, S. Kahan, R. Korry, C. McCann, and B. J. Smith, "Scheduling on the Tera MTA," in *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1995, pp. 19–44.

[9] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, March 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[10] C. Caşcaval, J. G. C. nos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. W. Jr., “Evaluation of a multithreaded architecture for cellular computing,” in *HPCA*, 2002, pp. 311–322.

[11] G. Alm$\acute{a}$i, C. Cascaval, J. G. Casta$\tilde{n}$os, M. Denneau, D. Lieber, Jos$\acute{e}$ E. Moreira, and J. Henry S. Warren, “Dissecting cyclops: a detailed analysis of a multithreaded architecture,” *SPECIAL ISSUE: MEDEA workshop*, vol. 31, pp. 26 – 38, 2003.

[12] G. S. Almasi, C. Caşcaval, J. E. Moreira, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, D. Newns, M. Snir, and J. Henry S. Warren, “Demonstrating the scalability of a molecular dynamics application on a petaflop computer,” in *ICS ’01: Proceedings of the 15th international conference on Supercomputing*. New York, NY, USA: ACM Press, 2001, pp. 393–406.

[13] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, , and D. J. Lipman, “A basic local alignment search tool,” *Journal of Molecular Biology*, vol. 215, p. 403C410, 1990.

[14] HMMER: sequence analysis using profile hidden Markov models. [Online]. Available: http://hmmer.wustl.edu/

[15] V. S. Pande, I. B. 1, J. Chapman, S. P. Elmer, S. Khaliq, S. M. Larson, Y. M. Rhee, M. R. Shirts, C. D. Snow, E. J. Sorin, and B. Zagrovic, “Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing,” *Biopolymers*, vol. 68, no. 1, pp. 91–109, 2003.

[16] B. M. E. Moret, D. A. Bader, and T. Warnow, “High-performance algorithm engineering for computational phylogenetics,” *J. Supercomput.*, vol. 22, no. 1, pp. 99–111, 2002.

[17] C. Laurent, F. Peyrin, J.-M. Chassery, and M. Amiel, “Parallel image reconstruction on MIMD computers for three-dimensional cone-beam tomography,” *Parallel Computing*, vol. 24, no. 9-10, pp. 1461–1479, 1998.

[18] S. K. Warfield, F. A. Jolesz, and R. Kikinis, “A high performance computing approach to the registration of medical imaging data,” *Parallel Computing*, vol. 24, no. 9-10, pp. 1345–1368, 1998.

[19] G. E. Christensen, “MIMD vs. SIMD parallel processing: A case study in 3d medical image registration.” *Parallel Computing*, vol. 24, no. 9-10, pp. 1369–1383, 1998.

[20] M. Stacy, D. P. Hanson, J. J. Camp, and R. A. Robb, “High performance computing in biomedical imaging research,” *Parallel Computing*, vol. 24, no. 9-10, pp. 1287–1321, 1998.

[21] Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/

[22] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. on Computers*, vol. 21, no. 9, pp. 948–960, Sep. 1972.

[23] Barry Wilkinson and Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, 1st ed. Upper Saddle River, New Jersey: Prentice Hall, August 12 1998.

[24] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture : A Hardware/-Software Approach*, 1st ed. San Francisco, CA: Morgan Kaufmann, August 1 1998.

[25] N. Adiga, M. Blumrich, and T. Liebsch, "An overview of the BlueGene/L supercomputer," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Baltimore, Maryland, 2002, pp. 1 – 22.

[26] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[27] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambrdge, MA: MIT Press, Oct. 1994.

[28] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. R. Rattner, "Platform 2015: Intel processor and platform evolution for the next decade," *Technology at Intel Magazine*, 2005. [Online]. Available: http://www.intel.com/technology/magazine/computing/platform-2015-0305.htm

[29] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996.

[30] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995.

[31] D. R. Butenhof, *Programming with POSIX(R) Threads*. Addison-Wesley Pub. Co., 1997.

[32] P. Thulasiraman, "Irregular computations on fine-grain multithreaded architecture," Ph.D. dissertation, University of Delaware, Newark, DE, 2000.

[33] J. B. Dennis and G. R. Gao, "Multithreaded architectures: Principles, projects, and issues," in *Multithreaded Computer Architecture: A Summary of the State of the Art*, R. A. Iannucci, G. R. Gao, R. H. Halstead, Jr., and B. Smith, Eds. Norwell,

Mass.: Kluwer Academic Pub., 1994, ch. 1, pp. 1–72, book contains papers presented at the Workshop on Multithreaded Computers, Albuquerque, N. Mex., Nov. 1991.

[34] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture," in *Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the 32nd Annual Interantional Symposium on Computer Architecture (ISCA'05)*, Madison, Wisconsin, June 4 2005.

[35] ——, "Tiny threads: a thread virtual machine for the cyclops64 cellular architecture," in *Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th International Parallel and Distributed Processing System*, Denver, Colorado, April 3 - 8 2005.

[36] J.-L. Gaudiot and L. Bic, Eds., *Advanced Topics in Data-Flow Computing*. Englewood Cliffs, N. Jer.: Prentice-Hall, 1991, book contains papers presented at the First Workshop on Data-Flow Computing, Eilat, Israel, May 1989.

[37] J. von Neumann, "First draft of a report on the EDVAC," *IEEE Ann. Hist. Comput.*, vol. 15, no. 4, pp. 27–75, 1993.

[38] R. A. Iannucci, "Toward a dataflow/von Neumann hybrid architecture," in *Proc. of the 15th Ann. Intl. Symp. on Computer Architecture*, Honolulu, Haw., May–Jun. 1988, pp. 131–140.

[39] Platform 2015 Unveiled at IDF Spring 2005. [Online]. Available: http://www.intel.com/technology/techresearch/idf/platform-2015-keynote.htm

[40] J. Tisdall, *Beginning Perl for Bioinformatics*, 1st ed. Sebastopol, CA: O'Reilly, October 15 2001.

[41] The NCBI genebank statistics. [Online]. Available: http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html

[42] NCBI blast. [Online]. Available: http://www.ncbi.nlm.nih.gov/BLAST/

[43] M. Cameron, H. E. Williams, and A. Cannane, "Improved gapped alignment in blast," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 1, no. 3, pp. 116–129, 2004.

[44] C. A. Stewart, D. Hart, D. K. Berry, G. J. Olsen, E. A. Wernert, and W. Fischer, "Parallel implementation and performance of fastDNAml: a program for maximum likelihood phylogenetic inference," in *Supercomputing '01: Proceedings of the*

2001 ACM/IEEE conference on Supercomputing (CDROM). New York, NY, USA: ACM Press, 2001, pp. 20–30.

[45] B. M. Moret, D. Bader, T. Warnow, S. Wyman, and M. Yan, "GRAPPA: a high-performance computational tool for phylogeny reconstruction from gene-order data," in *Botany 2001*, Albuquerque, NM, August 12-16 2001.

[46] D. A. Bader, "Computational biology and high-performance computing," *Commun. ACM*, vol. 47, no. 11, pp. 34–41, 2004.

[47] W. Zhu, Y. Niu, J. Lu, C. Shen, and G. R. Gao, "A cluster-based solution for high performance hmmpfam using earth execution model," in *proceedings of Cluster 2003*, Hong Kong. P.R.China, December 1-4 2003, accepted to be published in a Special Issue of the International Journal of High Performance Computing and Networking (IJHPCN).

[48] Y. Niu, Z. Hu, and G. R. Gao, "Parallel reconstruction for parallel imaging space rip on cellular computer architecture," in *Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, Cambridge, MA, November 9-11, 2004.

[49] Y. Niu and K. E. Barner, "Generation of tactile graphics with multilevel digital halftoning," *IEEE transactions on Neural Systems and Rehabilitation Engineering*, Oct 2004, submitted.

[50] D. W. Mount, *Bioinformatics: Sequence and Genome Analysis*, 1st ed. Cold Spring Harbor Laboratory Press, March 15 2001.

[51] J. T. L. Wang, Q. heng Ma, and C. H. Wu, "Application of neural network to biological data mining: A case study in protein sequence classification," *Proceedings of KDD-2000*, pp. 305–309, 2000.

[52] T. K. Attwood, M. E. Beck, D. R. Flower, P. Scordis, and J. N. Selley, "The PRINTS protein fingerprint database in its fifth year," *Nucleic Acids Research*, vol. 26, no. 1, p. 304C308, 1998.

[53] S. Eddy, "Profile hidden markov models," *Bioinformatics*, vol. 14, pp. 755–763, 1998.

[54] A. Krogh, M. Brown, I. Mian, K. Sjolander, and D. Haussler, "Hidden markov models in computational biology: applications to protein modeling," *Journal of Molecular Biology*, vol. 235, pp. 1501–1531, 1994.

[55] P. Baldi, Y. Chauvin, T. Hunkapiller, and M. A. McClure, "Hidden markov models of biologicalprimary sequence information," *PNAS*, vol. 91, no. 3, pp. 1059–1063, 1994.

[56] S.E.Levinson, L.R.Rabiner, and M.M.Sondhi, "An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition," *Bell Syst.Tech.J.*, vol. 62, pp. 1035–1074, 1983.

[57] P. Baldi and S. Brunak, *Bioinformatics: The Machine Learning Approach*, 2nd ed. Cambridge, MA: The MIT Press, August 1 2001.

[58] A. Bateman, E. Birney, L. Cerruti, R. Durbin, L. Etwiller, S.R. Eddy, S. Griffiths-Jones, K.L. Howe, M. Marshall, and E.L.L. Sonnhammer, "The pfam protein families database," *Nucleic Acids Research*, vol. 30, no. 1, pp. 276–280, 2002.

[59] E.L.L Sonnhameer, S.R. Eddy, and R. Durbin, "Pfam: A comprehensive database of protein domain families based on seed alignments," *Proteins*, vol. 28, pp. 405–420, 1997.

[60] E.L.L Sonnhameer, S.R. Eddy, E. Birney, A. Bateman, and R. Durbin, "Pfam: Multiple sequence alignments and hmm-profiles of protein domains," *Nucl. Acids Res.*, vol. 26, pp. 320–322, 1998.

[61] G. Tremblay, K. B. Theobald, C. J. Morrone, M. D. Butala, J. N. Amaral, and G. R. Gao, "Threaded-c language reference manual (release 2.0)," *CAPSL Technical Memo 39*, 2000.

[62] C. J. Morrone, "An EARTH runtime system for multi-processor/multi-node Beowulf clusters," Master's thesis, Univ. of Delaware, Newark, DE, May 2001.

[63] C. Li, "EARTH-SMP: Multithreading support on an SMP cluster," Master's thesis, Univ. of Delaware, Newark, DE, Apr. 1999.

[64] C. Shen, "A portable runtime system and its derivation for the hardware SU implementation," Master's thesis, Univ. of Delaware, Newark, DE, December 2003.

[65] H. H. J. Hum, K. B. Theobald, and G. R. Gao, "Building multithreaded architectures with off-the-shelf microprocessors," *In Proceedings of the 8th International Parallel Processing Symposium*, pp. 288–294, 1994.

[66] The Argonne Scalable Cluster. [Online]. Available: http://www-unix.mcs.anl.gov/chiba/

[67] The Argonne JAZZ Cluster, Laboratory Computing Resource Center (LCRC). [Online]. Available: http://www.lcrc.anl.gov/jazz/

[68] 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*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[69] J. B. del Cuvillo, Z. Hu, W. Zhu, F. Chen, and G. R. Gao, "CAPSL memo 55: Toward a software infrastructure for the cyclops64 cellular architecture," CAPSL Group, Department of ECE, University of Delaware, Tech. Rep., 2004.

[70] D. W. McRobbie, E. A. Moore, M. J. Graves, and M. R. Prince, *MRI from Picture to Proton*. Cambridge University Press, December 5 2002.

[71] R. B. Buxton, *Introduction to Functional Magnetic Resonance Imaging : Principles and Techniques*. Cambridge University Press, November 2001.

[72] P. Woodward, *MRI for Technologists*. McGraw-Hill Companies, October 2000.

[73] Pruessmann KP, Weiger M, Boernert P, and Boesiger P, "SENSE, sensitivity encoding for fast MRI," *Magn Reson Med*, vol. 42, pp. 952–962, 1999.

[74] D. Atkinson, "Parallel imaging reconstruction," in *Workshop on Image Processing for MRI, held in conjunction with The 2004 British Chapter ISMRM meeting*, Edinburgh, UK, September 2004.

[75] Sodickson DK and Manning WJ, "Simultaneous acquistion of spatial harmonics(SMASH): fast imaging with radiofrequency coil arrays," *Magn Reson Med*, vol. 38, no. 4, pp. 591–603, 1997.

[76] Kyriakos WE, Panych LP, Kacher DF, Westin C-F, Bao SM, Mulkern RV, and Jolesz FA, "Sensitivity profiles from an array of coils for encoding and reconstruction in parallel (SPACE RIP)," *Magn Reson Med*, vol. 44, no. 2, pp. 301–308, 2000.

[77] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[78] G. R. Gao and S. J. Thomas, "An optimal parallel jacobi-like solution method for the singular value decomposition," in *Proc. Internat. Conf. Parallel Proc.*, 1988, pp. 47–53.

[79] M. Gu, J. Demmel, and I. Dhillon, "Efficient computation of the singular value decomposition with applications to least squares problems," Computer Science Dept., University of Tennessee, Knoxville, Tech. Rep. CS-94-257, 1994, LAPACK Working Note 88, http://www.netlib.org/lapack/lawns/lawn88.ps.

[80] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[81] J. Demmel, *Applied Numerical Linear Algebra*. SIAM, 1997.

[82] G. Golub and C. V. Loan, *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 1996.

[83] J. Demmel, M. Gu, S. Eisenstat, I. Slapnicar, K. Veselic, and Z. Drmac, "Computing the singular value decomposition with high relative accuracy," *Linear Algebra Appl.*, vol. 299, pp. 21–80, 1999.

[84] J. Demmel, "Accurate SVDs of structured matrices," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 3, pp. 562–580, 2000.

[85] J. Demmel and K. Veselic, "Jacobi's method is more accurate than QR," October 1989, lapack Working Note 15 (LAWN-15), Available from netlib, http://www.netlib.org/lapack/.

[86] M. R. Hestenes, "Inversion of matrices of biorthogonalization and related results," *J. Soc. Induct. Appl. Math.*, vol. 6, pp. 51–90, 1958.

[87] H. Rutishauser, "The jacobi method for real symmetric matrices," in *J. H. Wilkinson and C. Reinsch, editors, Linear Algebra, Volumn II of Handbook for Automatic Computations, chapter II/1*, vol. II, 1971, pp. 202–211.

[88] H. Park, "A real algorithm for the hermitian eigenvalue decomposition," *BIT*, vol. 33, pp. 158–171, 1993.

[89] G. E. Forsythe and P. Henrici, "The cyclic jacobi method for computing the principal values of a complex matrix," *Transactions of the American Methematical Society*, vol. 94, no. 1, pp. 1–23, 1960.

[90] R. P. Brent and F. T. Luk, "The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, no. 1, pp. 69–84, January 1985.

[91] R. P. Brent, F. T. Luk, and Charles F. Van Loan, "Computation of the singular value decomposition using mesh-connected processors," *Journal of VLSI and Computer Systems*, vol. 1, no. 3, pp. 242–260, 1985.

[92] D. Royo, M. Valero-García, and Antonio González, "Implementing the one-sided jacobi method on a 2D/3D mesh multicomputer," *Parallel Computing*, vol. 27, no. 9, pp. 1253–1271, August 2001.

[93] E. R. Hansen, "On cyclic jacoib methods," *Journal of Soc. Indust. Appl. Math.*, vol. 11, pp. 448–459, 1963.

[94] J. H. Wilkinson, *The Algebraic Eigenvalue Problem, pp. 277-278*. Oxford: Clarendon Press, 1965.

[95] S. Coren and L. Ward, *Sensation and Perception (3rd Edition)*. San Diego, California: Harcout Brace Jovanovich, 1989.

[96] ALVA Access Group Homepage. [Online]. Available: http://www.aagi.com/

[97] B. Lowenfield, *The Changing Status of the Blind: From Separations to Integration*. Springfield, Illinois: Charles C. Thomas, 1975.

[98] J. Bliss, "A relatively high-resolution reading aid for the blind," *IEEE Transactions on Man-Machine Systems*, vol. 10, no. 1, pp. 1–9, 1969.

[99] M. Kurze, L. Reichert, and T. Strothotte, "Access to business graphics for blind people," in *Proceedings of the RESNA'94 Annual conference*, Nashville, Tennessee, June 17-22 1994.

[100] T. P. Way and K. E. Barner, "Automatic visual to tactile translation, part I: Human factors, access methods and image manipulation," *IEEE Transactions on Rehabilitation Engineering*, vol. 5, no. 1, pp. 81–94, March 1997.

[101] ——, "Automatic visual to tactile translation, part II: Evaluation of the tactile image creation system," *IEEE Transactions on Rehabilitation Engineering*, vol. 5, no. 1, pp. 95–105, March 1997.

[102] S. Hernandez and K. E. Barner, "Joint region merging criteria for watershed-based image segmentation," in *International Conference on Image Processing*, Vancouver, BC, Canada, September 10-13 2000.

[103] ——, "Tactile imaging using watershed-based image segmentation," in *Proceedings of the fourth international ACM conference on Assistive technologies*, Arlington, Virginia, November 13-15 2000, pp. 26 – 33.

[104] S. Hernandez, K. E. Barner, and Y. Yuan, "Region merging using region homogeneity and edge integrity for watershed-based image segmentation," *Optical Engineering*, July 2004, accepted for publication.

[105] Repro-Tronics Homepage. [Online]. Available: http://www.repro-tronics.com/

[106] A. Nayak and K. E. Barner, "Optimal halftoning for tactile imaging," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, vol. 12, no. 2, pp. 216–227, June 2004.

[107] R. A. Ulichney, *Digital Halftoning*. Cambridge, MA: MIT Press, 1975.

[108] D. L. Lau, *Modern Digital Halftoning*. New York, NY: Marcel Dekker, 2000.

[109] H. R. Kang, *Digital Color Halftoning*. New York, NY: IEEE Press, 2000.

[110] B. E. Bayer, "An optimum method for two level rendition of continuous-tone pictures," in *IEEE International Conference on Communications*, Seattle, Washington, June 11-13 1973, pp. 11–15.

[111] R. W. Floyd and L. Steinberg, "An adaptive algorithm for spatial gray-scale," in *Proceedings Society Information Display*, vol. 17, February 1976, pp. 75–77.

[112] R. Levien, "Output dependant feedback in error diffusion halftoning," in *IS&T's Eighth International Congress on Advances in Non-Impact Printing Technologies*, Willianmsburg, Virginia, October 25-30 1992, pp. 280–282.

[113] American Thermoform Corporation Homepage. [Online]. Available: http://www.atcbrleqp.com/swell.htm

[114] Pulse Data International Homepage. [Online]. Available: http://www.pulsedata.co.nz

[115] ViewPlus Technologies. [Online]. Available: http://www.viewplustech.com

[116] S. Ellwanger and K. E. Barner, "Tactile image conversion and printing," *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, August 2004, submitted for publication.

[117] K. O. Johnson and J. R. Phillips, "Tactile spatial resolution: Two-point discrimination, gap detection, grating resolution, and letter recognition," *Journal of Neurophysiology*, vol. 46, no. 6, 1981.

[118] J. Sullivan, L. Ray, and R. Miller, "Design of minimum visual modulation halftone patterns," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, no. 1, pp. 33–38, Jan/Feb 1991.

[119] M. Rodriguez, "Graphic arts perspective on digital halftoning," in *Proceedings of SPIE, Human Vision, Visual Processing and Digital Display V*, B. E. Rogowitz and J. P. Allebach, Eds., vol. 2179, Feb 1994, pp. 144–149.

[120] J. F. Jarvis, C. N. Judice, and W. H. Ninke, "A survey of technique for the display of continuous-tone pictures on bilevel displays," *Computer Graphics and Image Processing*, vol. 5, no. 1, pp. 13–40, 1976.

[121] P. Stucki, “Mecca-a multiple-error correcting computation algorithm for bilevel image hardcopy reproduction,” RZ 1060 IBM Research Laboratory, Zurich, Switzerland, Tech. Rep., 1981.

[122] T. S. Hang, “PCM picture transmission,” *IEEE Spectrum*, vol. 2, no. 12, pp. 57–63, 1965.

[123] R. S. Gentile, E. Walowit, and J. P. Allebach, “Quantization and multilevel halftoning of color images for near original image quality,” *Journal of the Optical Society of America A*, vol. 7, no. 6, pp. 1019–1026, 1990.

[124] K.E.Spaulding, R.L.Miller, and J. Schildkraut, “Methods for generating blue-noise dither matrices for digital halftoning,” *Journal of Electronic Imaging*, vol. 6, no. 3, pp. 208–230, 1997.

[125] F. Faheem, D. L. Lau, and G. R. Arce, “Digital multitoning using gray level separation,” *The Journal of Imaging Science and Technology*, vol. 46, no. 5, pp. 385–397, September 2002.

[126] Microsoft Corporation, Ed., *Microsoft Windows 2000 Driver Development Kit*. Microsoft Press, April 2000.

[127] B. Lowenfeld, “Effects of blindness on the cognitive functions of children,” in *Berthold Lowefeld on Blindness and Blind People*, B. Lowenfeld, Ed. American Foundation for the Blind, 1981.

[128] T. P. Way, “Automatic generation of tactile graphics,” Master’s thesis, University of Delaware, Newark, DE, Fall 1996.