MULTITHREADING AND VECTORIZING STRATEGIES FOR CLUSTALW ON MULTICORE ARCHITECTURE

KRIDSADAKORN CHAICHOOMPU

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
MASTER OF ENGINEERING IN COMPUTER ENGINEERING
SCHOOL OF GRADUATE STUDIES
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
2006

ISBN 974-15-2799-3

หัวข้อวิทยานิพนธ์	กลวิธีมัลติเธรดดิ้งและเว็คเตอไรซิ่งสำหรับโปรแกรม ClustalW บนเครื่องมัลติคอร์
นักศึกษา	กฤษดากร ไชยชุมภู
รหัสนักศึกษา	47060809
ปริญญา	วิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
พ.ศ.	2549
อาจารย์ผู้ควบคุมวิทยานิพนธ์	ผศ. ดร. สุรินทร์ กิตติธรกุล

บทคัดย่อ

วิธีการจัดตำแหน่งข้อมูลของโปรตีนหรือดีเอ็นเอจำนวนมากๆ (sequence alignment of multiple protein or DNA) เป็นกระบวนการหนึ่งที่สำคัญในงานทางด้านไบโออินฟอร์เมติกส์ ซึ่งเครื่องมือประเภทนี้ที่ใช้กันแพร่หลายมากที่สุดคือโปรแกรมที่ชื่อว่า ClustalW โดยประกอบไปด้วยอัลกอริธึม 3 ส่วน คือ การจับคู่เพื่อจัดตำแหน่งข้อมูล (pairwise alignment) การสร้างไกด์ทรี (guide tree generation) และการจัดตำแหน่งแบบโพรเกรสซีพ (progressive alignment) งานวิจัยนี้ได้เพิ่มประสิทธิภาพโปรแกรม ClustalW แบบหนึ่งที่มีชื่อว่า ClustalW-SMP ซึ่งเป็นโปรแกรมแบบมัลติเธรดแต่ไม่สมบูรณ์แบบ และเสนอแนะวิธีการที่ช่วยคอมไพล์เลอร์เพื่อออฟติไมซ์โปรแกรมได้อย่างมีประสิทธิภาพมากขึ้น โดยเป้าหมายเพื่อที่จะทำให้โปรแกรม ClustalW ทำงานแบบมัลติเธรดได้อย่างสมบูรณ์ อีกทั้งทำให้ได้ผลลัพธ์รวดเร็วมากขึ้น ซึ่งรองรับการทำงานบนเครื่องแบบมัลติคอร์ โดยสรุปแล้วโปรแกรม ClustalW ที่ทำการปรับปรุงเรียบร้อยแล้ว สามารถทำงานได้อย่างเต็มประสิทธิภาพและรวดเร็ว

Thesis Title	Multithreading and Vectorizing Strategies for ClustalW on Multicore Architecture
Student	Kridsadakorn Chaichoompu
Student ID.	47060809
Degree	Master of Engineering
Program	Computer Engineering
Year	2006
Thesis Advisor	Asst. Prof. Dr. Surin Kittitornkun

ABSTRACT

Sequence alignment of multiple protein or DNA is an important tool in bioinformatics. The most widely used tool for aligning multiple protein or nucleotide sequences called ClustalW, which consists of three stages: pairwise alignment, guide tree generation and progressive alignment. This thesis enhances a multithreaded implementation of ClustalW called ClustalW-SMP and proposes the methodology that assists the compiler to optimize ClustalW for higher throughput. Our goal is to maximize the degree of parallelism and the throughput of execution on multithreading ClustalW and targets on multicore architecture. As a result, multithreading ClustalW with optimization is able to fully utilize the machine resources and gains higher throughput on multicore machines.

Acknowledgements

First of all, I would like to thank Assistant Professor Doctor Surin Kittitornkun, my Advisor, for his many suggestions and constant support during this research. I am also thankful to Professors in Department of Computer Engineering for their constructive comments and helpful discussions which gave me a better perspective on my own results. Finally, I would like to acknowledge the support of Dr. Sissades Tongsima, my family and my friends for all of their help.

Bangkok, Thailand Kridsadakorn Chaichoompu

Oct, 2006

Contents

Contents (con.)

Page

Contents (con.)

List of Tables

List of Figures

List of Figures (con.)

Chapter 1

Introduction

1.1 Motivation

Bioinformatics and computational biology involve with the use of techniques from applied mathematics, informatics, statistics, and computer science to solve biological problems. Research in computational biology often overlaps with systems biology. Major research efforts in the field include sequence alignment, gene finding, genome assembly, protein structure alignment, protein structure prediction, prediction of gene expression and protein-protein interactions, and the modeling of evolution.

The terms bioinformatics and computational biology are often used interchangeably, although the former typically focuses on algorithm development and specific computational methods, while the latter focuses more on hypothesis testing and discovery in the biological domain. Although this distinction is used by National Institutes of Health, USA in their working definitions of Bioinformatics and Computational Biology, it is clear that there is a tight coupling of developments and knowledge between the more hypothesis-driven research in computational biology and technique-driven research in bioinformatics. Computational biology also includes lesser known but equally important subdisciplines such as computational biochemistry and computational biophysics.

A common thread in projects in bioinformatics and computational biology is the use of mathematical tools to extract useful information from noisy data produced by high-throughput biological techniques such as genomics (The field of data mining overlaps with computational biology in this regard). A representative problem in bioinformatics is the assembly of high-quality DNA sequences from fragmentary "shotgun" DNA sequencing, while in computational biology; a representative problem might be statistical testing of a hypothesis of common gene regulation using data from mRNA microarrays or mass spectrometry.

1.2 Existing Approaches

Multiple alignments of protein sequences are important in many applications, including phylogenetic tree estimation, structure prediction and critical residue identication. Many Multiple Sequence Alignment (MSA) tools such as MSA [1] and PRALINE [2] have been proposed to reduce the high computation time of fully performing alignment of all sequences. ClustalW particularly is the most popular sequential program for multiple sequence alignment, and CLUSTALX [3] is a graphical interface version of ClustalW.

Overview of parallel processing, a fast implementation of the Smith-Waterman sequence-alignment algorithm using Single-Instruction, Multiple-Data (SIMD) technology is presented. This implementation is based on the MultiMedia eXtensions (MMX) and Streaming SIMD Extensions (SSE) technology that is embedded in Intel's latest microprocessors. Six-fold speedup relative to the fastest previously known Smith-Waterman implementaton on the same hardware was achieved by an optimized 8-way parallel processing approach. [4] More about Smith-Waterman algorithm, a preliminary implementation of Smith-Waterman algorithm using a new chip multiprocessor architecture with multiple Digital Signal Processors (DSP) on a single chip leading to high performance at low cost. [5] Several methods of computing sequence alignments with limited memory per processing element on SIMD processing elements were presentd in [6]. And this paper [7] improves alignment times by either reducing the alignment sensitivity or by developing specialized hardware. The solution comes in the form of parallel processing hardware such as Paracels GeneMatcher and Compugens Bioccelerator.

The parallel version of ClustalW is presented by SGI (Silicon Graphics Inc.) [8]. The SGI parallel version shows speedup of up to 10 folds when running ClustalW on 16 CPUs [9]. pCLUSTAL presents a parallel version of CLUSTALW. In contrast to the commercial SGI parallel Clustal version, which requires an expensive SGI multiprocessor system, pCLUSTAL can run on PC clusters as well [10]. ClustalW-MPI [11] is another interesting parallel ClustalW which uses a message-passing library called MPI and runs on distributed workstation clusters. Moreover, the parallel ClustalW with Dynamic Scheduling [12] proposes the algorithm that divides a progressive

alignment into subtasks and schedules them dynamically. Besides the software approach, a new approach MSA on reconfigurable hardware platforms to gain high performance at low cost. Fine-grained parallel processing elements (PEs) are designed for the computation of pairwise distances between protein sequences. [13] Caching and parallel methods are focused to improve the computation for the better throughput. The efficient execution of MSA method is concerned in the data server and the cluster of workstations about the effect of data caching. [14], [15]

1.3 Statement of Problem

In the near future, the multicore workstations will be commonly used even home or office. Instead of using the single CPU machines, the multicore workstations will serve the jobs for the faster execution time. But nowadays ClustalW, multiple sequence alignment tool, which is the sequential applications must be used and produce the low throughput. Therefore we have to waste the time for waiting the results especially the application that deals with the enormous data. This thesis presents the multithreading strategies and the optimizing methodologies to improve the sequential ClustalW for the higher throughput on the multicore workstations.

1.4 Contributions

This thesis proposes and applies the strategies to make use of multicore workstations. This thesis contributes the following:

A. It analyzes the sequential MSA to show the bottle neck of the application and the list of top usage functions.
B. It suggests the multithreading strategies for improving the sequential applications to increase throughput.
C. It suggests the optimizing and vectorizing methodologies for compact code and faster execution.

1.5 The Organization of this Thesis

This thesis is organized in the following manner. Chapter 2 reviews the background for the work presented in this thesis. Chaper 3 presents the theoretical backgrounds and our proposed multithreading strategy and optimizing and vectorizing methodology. Then, Chapter 4 describes the experiment and shows the results. Finally, Chapter 5 concludes this research work by providing the conclusion.

Chapter 2
Literature Review

2.1 Bioinformatics Applications

Bioinformatics is the use of Information Technology (IT) in biotechnology for the data storage, data warehousing and analyzing the DNA and protein sequences. In Bioinfomatics knowledge of many branches are required like biology, mathematics, computer science, laws of physics & chemistry, and of course sound knowledge of IT to analyze biotech data. Bioinformatics is not limited to the computing data, but in reality it can be used to solve many biological problems and find out how living things work.

It is the comprehensive application of mathematics (e.g., probability and statistics), science (e.g., biochemistry), and a core set of problem-solving methods (e.g., computer algorithms) to the understanding of living systems.

2.1.1 Sequence Alignment

A sequence alignment in bioinformatics is a way of arranging DNA, RNA, or protein primary sequences to emphasize their regions of similarity, which may indicate functional or evolutionary relationships between the genes or proteins in the query. Sequences are typically written with their characters (generally amino acids or nucleotides) in aligned columns into which gaps are inserted so that successive columns contain identical or similar characters.

```
AAB24882        TYHMCQFHCRYVNNHSGEKLYECNERSKAFSCPSHLQCHKRRQIGEKTHEHNQCGKAFPT 60
AAB24881        --------------------YECNQCGKAFAQHSSLKCHYRTHIGEKPYECNQCGKAFSK 40
                                    ****: .***:  * *:** * :****.:* *******..

AAB24882        PSHLQYHERTHTGEKPYECHQCGQAFKKCSLLQRHKRTHTGEKPYE-CNQCGKAFAQ- 116
AAB24881        HSHLQCHKRTHTGEKPYECNQCGKAFSQHGLLQRHKRTHTGEKPYMNVINMVKPLHNS 98
                 **** *:***********:***:**.: .***************    :  *.: :
```

Figure 2.1 A sequence alignment between two human zinc finger proteins identified by GenBank accession number

If two sequences in an alignment share a common ancestor, mismatches can be interpreted as point mutations and gaps as indels (that is, insertion or deletion mutations) introduced in one or both lineages in the time since they diverged from one another. In protein sequence alignment, the degree of similarity between amino acids occupying a particular position in the sequence can be interpreted as a rough measure of how conserved a particular region or sequence motif is among lineages. The absence of substitutions, or the presence of only very conservative substitutions (that is, the substitution of amino acids whose side chains have similar biochemical properties) in a particular region of the sequence, suggest that this region has structural or functional importance. Although DNA and RNA nucleotide bases are more similar to each other than to amino acids, the conservation of base pairing can indicate a similar functional or structural role. Sequence alignment can be used for non-biological sequences, such as identifying similarities in a series of letters and words present in human language.

Very short or very similar sequences can be aligned by hand; however, most interesting problems require the alignment of lengthy, highly variable or extremely numerous sequences that cannot be aligned solely by human effort. Instead, human knowledge is primarily applied in constructing algorithms to produce high-quality sequence alignments, and occasionally in adjusting the final results to reflect patterns that are difficult to represent algorithmically (especially in the case of nucleotide sequences). Computational approaches to sequence alignment generally fall into two categories: **global alignments** and **local alignments**. Calculating a global alignment is a form of global optimization that "forces" the alignment to span the entire length of all query sequences. By contrast, local alignments identify regions of similarity within long sequences that are often widely divergent overall. Local alignments are often preferable, but can be more difficult to calculate because of the additional challenge of identifying the regions of similarity. A variety of computational algorithms have been applied to the sequence alignment problem, including slow but formally optimizing methods like dynamic programming and efficient heuristic or probabilistic methods designed for large-scale database search.

2.1.2 Global and Local Alignments

```
Global FTFTALILLAVAV
       F--TAL-LLA-AV

Local  FTFTALILL-AVAV
       --FTAL-LLAAV--
```

Figure 2.2 Illustration of global and local alignments

Global alignments, which attempt to align every residue in every sequence, are most useful when the sequences in the query set are similar and of roughly equal size. (This does not mean global alignments cannot end in gaps.) A general global alignment technique is called the Needleman-Wunsch algorithm and is based on dynamic programming. Local alignments are more useful for dissimilar sequences that are suspected to contain regions of similarity or similar sequence motifs within their larger sequence context. The Smith-Waterman algorithm is a general local alignment method based on dynamic programming. With sufficiently similar sequences, there is no difference between local and global alignments.

Hybrid methods, known as semiglobal or "glocal" methods, attempt to find the best possible alignment that includes the start of one or the other sequence and the end of one or the other sequence. This can be especially useful when the downstream part of one sequence overlaps with the upstream part of the other sequence. In this case, neither global nor local alignment is entirely appropriate: a global alignment would attempt to force the alignment to extend beyond the region of overlap, while a local alignment might not fully cover the region of overlap.

2.1.3 Pairwise Alignment

Pairwise sequence alignment methods are concerned with finding the best-matching piecewise (local) or global alignments of two query sequences. Pairwise alignments can only be used between two sequences at a time, but they are efficient to calculate and are often used for methods that do not require extreme precision, such as

searching a database for sequences with high homology to a query. The three primary methods of producing pairwise alignments are dot-matrix methods, dynamic programming, and word methods; however, most multiple sequence alignment techniques can align only two sequences. Although each method has its individual strengths and weaknesses, all three methods have difficulty with highly repetitive sequences of low information content, especially where the number of repetitions may be different in the two sequences to be aligned.

2.1.4 Multiple Sequence Alignment

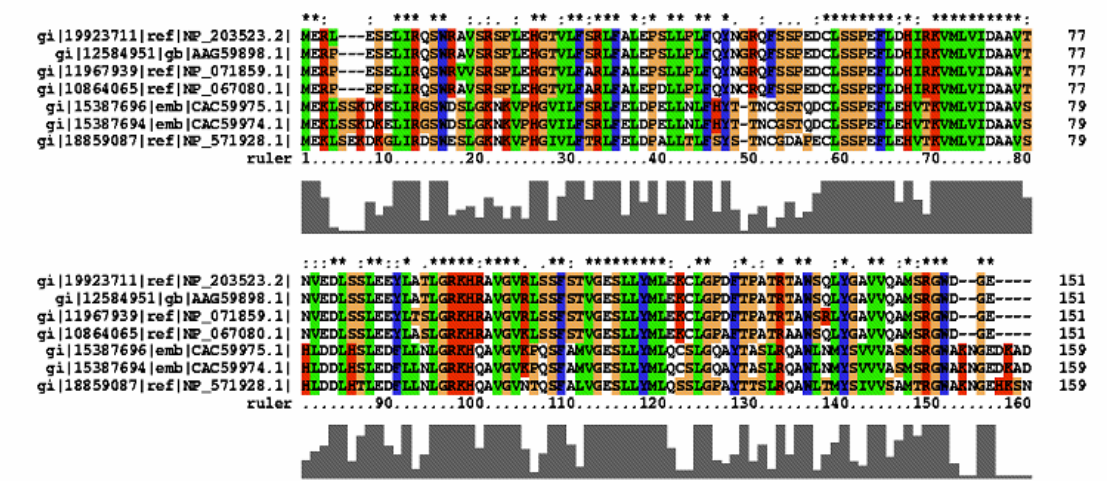

Figure 2.3 Multiple sequence alignment of 7 neuroglobins

Multiple sequence alignment (MSA) is an extension of pairwise alignment to incorporate more than two sequences at a time. Multiple alignment methods try to align all of the sequences in a given query set. Multiple alignments are often used in identifying conserved sequence regions across a group of sequences hypothesized to be evolutionarily related. Such conserved sequence motifs can be used in conjunction with structural and mechanistic information to locate the catalytic active sites of enzymes. Alignments are also used to aid in establishing evolutionary relationships by constructing phylogenetic trees. MSAs are computationally difficult to produce and most formulations of the problem lead to NP-complete combinatorial optimization

problems. Nevertheless, the utility of these alignments in bioinformatics has led to the development of a variety of methods suitable for aligning three or more sequences.

2.1.5 Progressive Methods

Progressive, hierarchical, or tree methods generate an MSA by first aligning the most similar sequences and then adding successively less related sequences or groups to the alignment until the entire query set has been incorporated into the solution. The initial tree describing the sequence relatedness is based on pairwise comparisons that may include heuristic pairwise alignment methods similar to FASTA. Progessive alignment results are dependent on the choice of "most related" sequences and thus can be sensitive to inaccuracies in the initial pairwise alignments. Most progressive MSA methods additionally weight the sequences in the query set according to their relatedness, which reduces the likelihood of making a poor choice of initial sequences and thus improves alignment accuracy.

Many variations of the Clustal progressive implementation are used for multiple sequence alignment, phylogenetic tree construction, and as input for protein structure prediction. A slower but more accurate variant of the progressive method is known as T-Coffee; implementations can be found at ClustalW and T-Coffee.

2.1.6 Case Study: ClustalW Software

ClustalW has become the most popular algorithm for multiple sequence alignment. This program implements a progressive method for multiple sequence alignment. As a progressive algorithm, ClustalW adds sequences one by one to the existing alignment to build a new alignment. The order of the sequences to be added to the new alignment is indicated by a precomputed phylogenetic tree, which is called a **guide tree**. The guide tree is constructed using the similarity of all possible pairs of sequences. The algorithm consists of 3 phases that are described below:

Stage 1: Distance Matrix: All pairs of sequences are aligned separately in order to calculate a distance matrix based on the percentage of mismatches of each pair of sequences.

Stage 2: Neighbor joining: The guide tree is calculated from the distance matrix using a neighbor joining algorithm [16]. The guide tree defines the order which the sequences are aligned in the next stage.

Stage 3: Progressive alignment: The sequences are progressively aligned according the guide tree.

2.2 Parellel Computing and Multithreading

2.2.1 Parallel Computing

Parallel computing is the simultaneous execution of the same task (split up and specially adapted) on multiple processors in order to obtain results faster. The idea is based on the fact that the process of solving a problem usually can be divided into smaller tasks, which may be carried out simultaneously with some coordination. A parallel computing system is a computer with more than one processor for parallel processing. The recent multicore processors are also parallel computing systems. There are many different kinds of parallel computers. They are distinguished by the kind of interconnection between processors (known as "processing elements" or PEs), processors and memories. Traditionally, Flynn's taxonomy classifies parallel (and serial) computers according to

A. SISD (single instruction stream, single data stream)
B. SIMD (single instruction stream, multiple data streams)
C. MISD (multiple instruction streams, single data stream)
D. MIMD (multiple instruction streams, multiple data streams)

One major way to classify parallel computers is based on their memory architectures. Shared memory parallel computers have multiple processors accessing all available memory as global address space. They can be further divided into two main classes based on memory access times: Uniform memory access (UMA), in which access times to all parts of memory are equal, or Non-Uniform memory access (NUMA),

in which they are not. Distributed memory parallel computers also have multiple processors, but each of the processors can only access its own local memory; no global memory address space exists across them. Parallel computing systems can also be categorized by the numbers of processors in them. Systems with thousands of such processors are known as massively parallel. Subsequently there is what is referred to as "Large scale" vs "Small scale" parallel processors. This depends on the size of the processor, eg. a PC based parallel system would generally be considered a small scale system.

Parallel processor machines are also divided into symmetric and asymmetric multiprocessors, depending on whether all the processors are the same or not (for instance if only one is capable of running the operating system code and others are less privileged). A variety of architectures have been developed for parallel processing. For example, Ring architecture has processors linked by a ring structure. Other architectures include Hypercubes, Fat trees, systolic arrays, and so on.

Approaches to parallel computers include:

A. Multiprocessing
B. Computer cluster
C. Parallel supercomputers
D. Distributed computing
E. NUMA vs. SMP vs. massively parallel computer systems
F. Grid computing

Some frequently used terms in parallel computing are:

A. Task: A logically high level, discrete, independent section of computational work. A task is typically executed by a processor as a program
B. Synchronization: The coordination of simultaneous tasks to ensure correctness and avoid unexpected race conditions.
C. Speedup: Also called parallel speedup, which is defined as wall-clock time of best serial execution divided by wall-clock time of parallel execution.

D. Parallel overhead: The extra work associated with parallel version compared to its sequential code, mostly the extra CPU time and memory space requirements from synchronization, data communications, parallel environment creation and cancellation, etc.
E. Scalability: A parallel system's ability to gain proportionate increase in parallel speedup with the addition of more processors.

A parallel programming model is a set of software technologies to express parallel algorithms and match applications with the underlying parallel systems. It encloses the areas of applications, languages, compilers, libraries, communication systems, and parallel I/O. People have to choose a proper parallel programming model or a form of mixture of them to develop their parallel applications on a particular platform.

Parallel models are implemented in several ways: as libraries invoked from traditional sequential languages, as language extensions, or complete new execution models. They are also roughly categorized for two kinds of systems: shared memory systems and distributed memory systems, though the lines between them are largely blurred nowadays.

2.2.2 Multithreading

Multithreading is a technique for improving the overall efficiency of superscalar CPUs. Multithreading permits multiple independent threads of execution to better utilize the resources provided by modern processor architectures. Normal multithreading operating systems allow multiple processes and threads to utilize the processor one at a time, giving exclusive ownership to a particular thread for a time slice in the order of milliseconds - this is called Temporal multithreading. Quite often, a process will stall for hundreds of cycles while waiting for some external resource (for example, a RAM load), thus lowering processor efficiency.

A successive improvement is super-threading, where the processor can execute instructions from a different thread each cycle. Thus cycles left unused by a thread can

be used by another that is ready to run. Still, a given thread is almost surely not utilizing all the multiple execution units of a modern processor at the same time. Simultaneous multithreading allows multiple threads to execute different instructions in the same clock cycle, using the execution units that the first thread left spare. This is done without great changes to the basic processor architecture: the main additions needed are the ability to fetch instructions from multiple threads in a cycle, and a larger register file to hold data from multiple threads. The number of concurrent threads can be decided by the chip designers, but practical restrictions on chip complexity usually limit the number to 2, 4 or sometimes 8 concurrent threads.

Since the technique is really an efficiency solution, and there is inevitable increased conflict on shared resources, measuring or agreeing on the "goodness" of the solution can be difficult. Some researchers have shown that the extra threads can be used to proactively seed a shared resource like a cache, to improve the performance of another single thread, and claim this shows that multithreading is not just an efficiency solution. Others use multithreading to provide redundant computation, for some level of error detection and recovery. But, in most current cases, multithreading is about efficiency and increased throughput of computations, per amount of hardware used.

In processor design, there are two ways to increase on-chip parallelism with less resource requirement: one is superscalar technique which tries to increase Instruction Level Parallelism (ILP), the other is multithreading approach exploiting Thread Level Parallelism (TLP). Superscalar means executing multiple instructions at the same time while chip-level multithreading (CMT) executes instructions from multiple threads within one processor chip at the same time. There are many ways to support more than one thread within a chip, namely:

A. Multithreaded: Interleaved issue of multiple instructions from different threads
B. Simultaneous multithreading (SMT): Issue multiple instructions from multiple threads in one cycle.
C. Chip-level multiprocessing (CMP or Multicore): integrate two or more superscalar processors into one chip, each execute one thread independently

D. Any combination of multithreaded/SMT/CMP

The key factor to distinguish them is to look at how many instructions the processor can issue in one cycle and how many threads from which the instructions come. For example, Sun Microsystems' UltraSPARC T1 (known as "Niagara" until its November 14, 2005 release) is a multicore processor combined with multithreaded technique instead of simultaneous multithreading because each core can only issue one instruction at a time.

The Intel Pentium 4 was the first modern desktop processor to implement simultaneous multithreading, starting from the 3.06GHz model released in 2002, and since introduced into a number of their processors. Intel calls the functionality Hyper-Threading Technology (HTT), and provides a basic two-thread SMT engine. Intel claims up to a 30% speed improvement compared against an otherwise identical, non-SMT Pentium 4. The performance improvement seen is very application dependent, however, and some programs actually slow down slightly when HTT is turned on. This is due to the replay system of the Pentium 4 tying up valuable execution resources, thereby starving the other thread. However, any performance degradation is unique to the Pentium 4 (due to various architectural nuances), and is not characteristic of SMT in general.

The latest MIPS architecture designs include a two-thread SMT system known as "MIPS MT". RMI, a Cupertino-based startup is the first MIPS vendor to provide a processor SOC based on 8 cores, each of which runs 4 threads. The threads can be run in fine-grain mode where a different thread can be executed each cycle. The threads can also be assigned priorities.

The IBM POWER5, announced in May 2004, is a dual core processor, with each core including a two-thread SMT engine. IBM's implementation is more sophisticated than the previous ones, because it can assign a different priority to the various threads, is more fine grained, and the SMT engine can be turned on and off dynamically, to better execute those workloads where a SMT processor would not increase performance. This is IBM's second implementation of generally available hardware multithreading.

Although many people reported that Sun Microsystems' UltraSPARC T1 (known as "Niagara" until its 14 November 2005 release) and the upcoming processor codenamed "Rock" are implementations of SPARC focused almost entirely on exploiting SMT and CMP techniques, Niagara is not actually using SMT. Sun refers to these combined approaches as "CMT", and the overall concept as "Throughput Computing". The Niagara chip uses fine-grained multithreading. Unlike SMT, where instructions from multiple threads can be issued simultaneously, the processor uses a round robin policy to issue instructions from a single thread each cycle. The designers of the Montecito (processor) have also chosen not to use SMT.

2.3 Multicore Processors

A multicore microprocessor is one which combines two or more independent processors into a single package, often a single integrated circuit (IC). A dual core device contains only two independent microprocessors. In general, multicore microprocessors allow a computing device to exhibit some form of thread-level parallelism (TLP) without including multiple microprocessors in separate physical packages. This form of TLP is often known as chip-level multiprocessing, or CMP.

There is some discrepancy in the semantics by which the terms "multicore" and "dual core" are defined. Most commonly they are used to refer to some sort of central processing unit (CPU). Additionally, some use these terms only to refer to multicore microprocessors that are manufactured on the same integrated circuit die. These persons generally prefer to refer to separate microprocessor dies in the same package by another name, such as "multi-chip module", "double core", or even "twin core". This thesis uses both the terms "multicore" and "dual core" to reference microelectronic CPUs manufactured on the same integrated circuit, unless otherwise noted.

2.3.1 Development Motivation

A. Technical pressures

While CMOS manufacturing technology continues to improve, reducing the size of single gates, physical limits of semiconductor-based microelectronics become a

major design concern. Some effects of these physical limitations can cause significant heat dissipation and data synchronization problems. The demand for more complex and capable microprocessors causes CPU designers to utilize various methods of increasing performance. Some ILP methods like superscalar pipelining are suitable for many applications, but are inefficient for others that tend to contain difficult-to-predict code. Many applications are better suited to TLP methods, and multiple independent CPUs are one common method used to increase a system's overall TLP. A combination of increased available space due to refined manufacturing processes and the demand for increased TLP led to the logical creation of multicore CPUs.

B. Commercial incentives

Several business motives drive the development of dual core architectures. Since SMP designs have been long implemented using discrete CPUs, the issues regarding implementing the architecture and supporting it in software are well known. Additionally, utilizing a proven processing core design (e.g. Freescale's e700 core) without architectural changes reduces design risk significantly. Finally, the connotations of the terminology “dual core” (and other multiples) lend itself to marketing efforts.

Additionally, for general-purpose processors, much of the motivation for multicore processors comes from the increasing difficulty of improving processor performance by increasing the operating frequency (frequency-scaling). In order to continue delivering regular performance improvements for general-purpose processors, manufacturers such as Intel and AMD have turned to multicore designs, sacrificing lower manufacturing costs for higher performance in some applications and systems.

Multicore architectures are being developed, but so are the alternatives. An especially strong contender for established markets is to integrate more peripheral functions into the chip.

2.3.2 Advantages

A. Proximity of multiple CPU cores on the same die have the advantage that the cache coherency circuitry can operate at a much higher clock rate than is possible if the signals have to travel off-chip, so combining equivalent CPUs on a single die

significantly improves the performance of cache snoop (alternative: Bus_snooping) operations.

B. Assuming that the die can fit into the package, physically, the multicore CPU designs require much less Printed Circuit Board (PCB) space than multi-chip SMP designs.

C. A dual core processor uses slightly less power than two coupled single-core processors, principally because of the increased power required to drive signals external to the chip and because the smaller silicon process geometry allows the cores to operate at lower voltages; furthermore, the cores share some circuitry, like the L2 cache and the interface to the front side bus (FSB).

D. In terms of competing technologies for the available silicon die areas, multicore design can make use of proven CPU core library designs and produce a product with lower risk of design error than devising a new wider core design. Also, adding more cache suffers from diminishing returns.

2.3.3 Disadvantages

A. Multicore processors do not need the operating system (OS) to support them, but instead adjustments to existing software are required to maximize the computing resources provided by multicore processors. Also, the ability of multicore processors to increase application performance depends on using threaded applications to optimize the use of resources.

B. Integration of a multicore chip drives production yields down and they are more difficult to manage thermally than lower-density single-chip designs.

C. From an architectural point of view, ultimately, single CPU designs may make better use of the silicon surface area than multiprocessing cores, so a development commitment to this architecture may carry the risk of obsolescence.

2.3.4 Software Impact

Most existing software is not ready to directly harness the power of multicore processors since they are written in traditional sequential programming languages like

C, C++ and FORTRAN, all of which have the limited scope of only one processor in mind. However, advances in software virtualization techniques such as running virtual machines will benefit from adoption of multiple core architectures by allowing users to more independantly co-execute existing software in a non-tandem fashion.

Current software titles that fully utilize multicore technologies include: Maya, Blender3D, Quake 3 & 4, Elder Scrolls: Oblivion, 3DS Max, Adobe Photoshop, Windows XP Professional, Windows 2003, Mac OS X, Linux, and many operating systems that are streamlined for server use.

Parallel programming is a must option for a single software to exploit multiple computation units (cores) simultaneously, often by multithread or multitask programming. Some existing parallel programming models such as OpenMP and MPI can be directly used on multicore platforms. Other research efforts have been seen also, like Cray's Chapel, Sun's Fortress, and IBM's X10.

Concurrency acquires a central role in true parallel application. The basic steps in designing parallel applications are:

A. Partitioning

The partitioning stage of a design is intended to expose opportunities for parallel execution. Hence, the focus is on defining a large number of small tasks in order to yield what is termed a fine-grained decomposition of a problem.

B. Communication

The tasks generated by a partition are intended to execute concurrently but cannot, in general, execute independently. The computation to be performed in one task will typically require data associated with another task. Data must then be transferred between tasks so as to allow computation to proceed. This information flow is specified in the communication phase of a design.

C. Agglomeration

In the third stage, we move from the abstract toward the concrete. We revisit decisions made in the partitioning and communication phases with a view to obtaining an algorithm that will execute efficiently on some class of parallel computer. In particular, we consider whether it is useful to combine, or agglomerate, tasks identified

by the partitioning phase, so as to provide a smaller number of tasks, each of greater size. We also determine whether it is worthwhile to replicate data and/or computation.

D. Mapping

In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute. This mapping problem does not arise on uniprocessors or on shared-memory computers that provide automatic task scheduling.

On the other hand, on the server side, multicore processors are ideal because they allow many users to connect to a site simultaneously and have independent threads of execution. This allows for Web servers and application servers that have much better throughput.

2.4 Compiler Optimization and Vectorizing

2.4.1 Compiler Optimization

Compiler optimization is the process of tuning the output of a compiler to minimise some attribute (or maximise the efficiency) of an executable program. The most common requirement is to minimise the time taken to execute a program, a less common one is to minimise the amount of memory occupied, and the growth of portable computers has created a market for minimising the power consumed by a program.

It has been shown that some code optimization problems are NP-complete. In practice, factors such as programmer willingness to wait for the compiler to complete its task place lower limits on the optimizations that a compiler implementor might provide (optimization is a very CPU and memory intensive process). In the past computer memory limitations were also a major factor in limiting which optimizations could be performed.

Compiler vendors often advertise their products as being optimizing compilers and the ability of a compiler to optimize code can affect its sales and the regard programmers have for it.

2.4.2 Types of Optimizations

Techniques in optimization can be broken up along various scopes which affect anywhere from a single statement to an entire program. Generally locally scoped techniques are easier to implement than global ones but result in lesser gains. Some examples of scopes include:

A. Peephole optimizations: Usually performed late in the compilation process after machine code has been generated. This form of optimization examines a few adjacent instructions (like looking through a peephole at the code) to see whether they can be replaced by a single instruction or a shorter sequence of instructions. For instance, a multiplication of a value by two might be more efficiently executed by shifting the value left or by adding the value itself (this example is also an instance of strength reduction).

B. Local or intraprocedural optimizations: These only consider information local to a function definition. This reduces the amount of analysis that needs to be performed (saving time and reducing storage requirements) but means that worst case assumptions have to be made when function calls occur or global variables are accessed (because little information about them is available).

C. Interprocedural or whole-program optimization: These analyse all of a programs source code. The greater quantity of information extracted means that optimizations can be more effective compared to when they only have access to local information (ie, within a single function). This kind of optimization can also allow new techniques to be performed. For instance function inlining, where a call to a function is replaced by a copy of the function body.

D. Loop optimizations: These acts on the statements which make up a loop, such as a for-loop (eg, loop-invariant code motion). Loop optimizations can have a significant impact because many programs spend a large percentage of their time inside loops.

In addition to scoped optimizations there are two further general categories of optimization:

A. Programming language-independent vs. language-dependent: Most high-level languages share common programming constructs and abstractions--decision (if,

switch, case), looping (for, while, repeat ... until, do ... while), encapsulation (structures, objects). Thus similar optimization techniques can be used across languages. However certain language features make some kinds of optimizations possible and/or difficult. For instance, the existence of pointers in C and C++ makes certain optimizations of array accesses difficult. Conversely, in some languages functions are not permitted to have "side effects". Therefore, if repeated calls to the same function with the same arguments are made, the compiler can immediately infer that results need only be computed once and the result referred to repeatedly.

B. Machine independent vs. machine dependent: Many optimizations that operate on abstract programming concepts (loops, objects, structures) are independent of the machine targeted by the compiler. But many of the most effective optimizations are those that best exploit special features of the target platform.

The following is an instance of a local machine dependent optimization. To set a register to 0, the obvious way is to use the constant 0 in an instruction that sets a register value to a constant. A less obvious way is to XOR a register with itself. It is up to the compiler to know which instruction variant to use. On many RISC machines, both instructions would be equally appropriate, since they would both be the same length and take the same time. On many other microprocessors such as the Intel x86 family, it turns out that the XOR variant is shorter and probably faster, as there will be no need to decode an immediate operand, nor use the internal "immediate operand register". (The catch being that XOR may introduce a data dependency on the previous value of the register, causing a pipeline stall that makes execution slower than using setting the register to a constant 0.)

2.4.3 Factors Affecting Optimization

A. The machine itself

Many of the choices about which optimizations can and should be done depend on the characteristics of the target machine. It is sometimes possible to parameterize some of these machine dependent factors, so that a single piece of compiler code can

be used to optimize different machines just by altering the machine description parameters. GCC is a compiler which exemplifies this approach.

B. The architecture of the target CPU

- Number of CPU registers: To a certain extent, the more registers, the easier it is to optimize for performance. Local variables can be allocated in the registers and not on the stack. Temporary/intermediate results can be left in registers without writing to and reading back from memory.
- RISC vs. CISC: CISC instruction sets often have variable instruction lengths, often have a larger number of possible instructions that can be used, and each instruction could take differing amounts of time. RISC instruction sets attempt to limit the variability in each of these: instruction sets are usually constant length, with few exceptions, there are usually fewer combinations of registers and memory operations, and the instruction issue rate (the number of instructions completed per time period, usually an integer multiple of the clock cycle) is usually constant in cases where memory latency is not a factor. There may be several ways of carrying out a certain task, with CISC usually offering more alternatives than RISC. Compilers have to know the relative costs among the various instructions and choose the best instruction sequence (see instruction selection).
- Pipelines: A pipeline is essentially an ALU broken up into an assembly line. It allows use of parts of the ALU for different instructions by breaking up the execution of instructions into various stages: instruction decode, address decode, memory fetch, register fetch, compute, register store, etc. One instruction could be in the register store stage, while another could be in the register fetch stage. Pipeline conflicts occur when an instruction in one stage of the pipeline depends on the result of another instruction ahead of it in the pipeline but not yet completed. Pipeline conflicts can lead to pipeline stalls: where the CPU wastes cycles waiting for a conflict to resolve.

Compilers can schedule, or reorder, instructions so that pipeline stalls occur less frequently.

C. Number of functional units: Some CPUs have several ALUs and FPUs. This allows them to execute multiple instructions simultaneously. There may be restrictions on which instructions can pair with which other instructions ("pairing" is the simultaneous execution of two or more instructions), and which functional unit can execute which instruction. They also have issues similar to pipeline conflicts.

Here again, instructions have to be scheduled so that the various functional units are fully fed with instructions to execute.

D. The architecture of the machine

- Cache size (256KB, 1MB) and type (fully associative, 4-way associative): Techniques like inline expansion and loop unrolling may increase the size of the generated code and reduce code locality. The program may slow down drastically if an oft-run piece of code (like inner loops in various algorithms) suddenly cannot fit in the cache. Also, caches which are not fully associative have higher chances of cache collisions even in an unfilled cache.
- Cache/Memory transfer rates: These give the compiler an indication of the penalty for cache misses. This is used mainly in specialized applications.

2.4.4 Optimization Methodologies

Some optimization techniques primarily designed to operate on loops include:

A. Induction variable analysis

Roughly, if a variable in a loop is a simple function of the index variable, such as j:=4*i+1, it can be updated appropriately each time the loop variable is changed. This is a strength reduction, and also may allow the index variable's definitions to become dead code. This information is also useful for bounds-checking elimination and dependence analysis, among other things.

B. Loop fission or loop distribution

Loop fission attempts to break a loop into multiple loops over the same index range but each taking only a part of the loop's body. This can improve locality of reference, both of the data being accessed in the loop and the code in the loop's body.

C. Loop fusion or loop combining

This technique is another technique which attempts to reduce loop overhead. When two adjacent loops would iterate the same number of times (whether or not that number is known at compile time), their bodies can be combined as long as they make no reference to each other's data.

D. Loop inversion

This technique changes a standard while loop into a do/while (also known as repeat/until) loop wrapped in an IF conditional, reducing the number of jumps by two, for cases when the loop is executed. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known at compile-time and is known to be side-effect-free, the IF guard can be skipped.

E. Loop interchange

These optimizations exchange inner loops with outer loops. When the loop variables index into an array, such a transformation can improve locality of reference, depending on the array's layout.

F. Loop-invariant code motion

If a quantity is computed inside a loop during iteration and its value is the same for each iteration. It can vastly improve efficiency to hoist it outside the loop and compute its value just once before the loop begins. This is particularly important with the address-calculation expressions generated by loops over arrays. For correct implementation, this technique must be used with loop inversion, because not all code is safe to be hoisted outside the loop.

G. Loop nest optimization

Some pervasive algorithms such as matrix multiplication have very poor cache behavior and excessive memory accesses. Loop nest optimization increases the number of cache hits by performing the operation over small blocks and by using a loop interchange.

H. Loop reversal

Loop reversal reverses the order in which values are assigned to the index variable. This is a subtle optimization which can help eliminate dependencies and thus enable other optimizations.

I. Loop unrolling

Duplicates the body of the loop multiple times, in order to decrease the number of times the loop condition is tested and the number of jumps, which hurt performance by impairing the instruction pipeline. Completely unrolling a loop eliminates all overhead, but requires that the number of iterations be known at compile time.

J. Loop splitting

Loop splitting attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. A useful special case is loop peeling, which can simplify a loop with a problematic first iteration by performing that iteration separately before entering the loop.

K. Loop unswitching

Unswitching moves a conditional inside a loop outside of it by duplicating the loop's body, and placing a version of it inside each of the if and else clauses of the conditional.

2.4.5 Vectorizing

Vectorization, in computer science, is the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector (series of adjacent values). Vector processing is a major feature of supercomputers since while there may be some overhead to starting up a vector operation, once it starts each individual operation is faster (in part because it avoids the need for instruction decoding).

One major research topic in computer science is the search for methods of automatic vectorization; seeking methods that would allow a compiler to convert scalar algorithms into vectorized algorithms without human assistance. More about vectorizing read in appendix B.

2.5 Related works

2.5.1 SGI version of ClustalW

The paper cited: Performance Optimization of Clustal W: Parallel Clustal W, HT Clustal, and MULTICLUSTAL[9]

This paper first outlines the efforts undertaken by SGI to parallelize the ClustalW application. The parallel version shows speedups of up to 10x when running Clustal W on 16 CPUs and significantly reduces the time required for data analysis. Second, the development of a high-throughput version of Clustal W called HT Clustal and the different methods of scheduling multiple MA jobs in HT Clustal are discussed. Third, improvements in the recently introduced MULTICLUSTAL algorithm and its efficient use of Parallel Clustal W are reviewed.

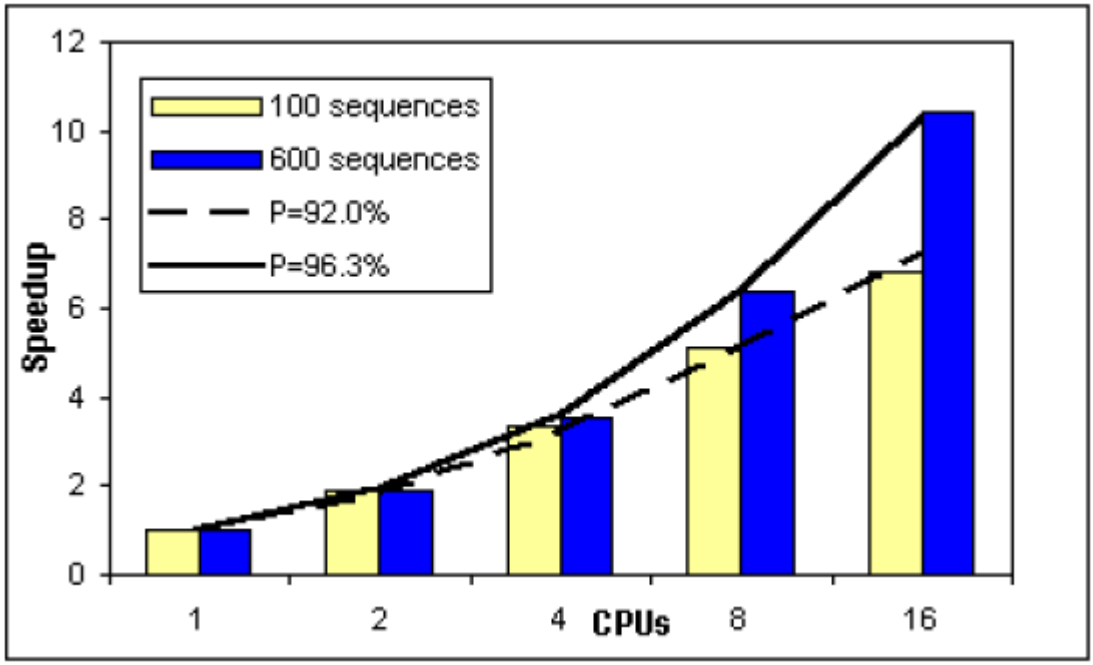

Figure 2.4 Experiment result of SGI version of ClustalW

Calculation was done for 100 and 600 GPCR protein sequences with the average length 390 amino acids. The speedup of more than 10x is seen for the MA of 600 GPCR proteins using 16 CPUs as compared to the uniprocessor time. Total time is

reduced from 1 hour, 7 minutes (uniprocessor) to just over 6.5 minutes (on 16 CPUs of the SGI Origin 3000 series).

2.5.2 pCLUSTALW

The paper cited: Parallel CLUSTAL W For PC Clusters [10]

This paper presents a parallel version of CLUSTALW, called pCLUSTAL. In contrast to the commercial SGI parallel Clustal, which requires an expensive shared memory SGI multiprocessor, pCLUSTAL can be run on a range of distributed and shared memory parallel machines, from high-end parallel multiprocessors to PC clusters. The implementation of pCLUSTAL uses C and the MPI communication library, and tests on a PC cluster. The experimental evaluation shows that our pCLUSTAL code achieves similar or better speedup on a distributed memory PC clusters than the commercial SGI parallel Clustal on a shared memory SGI multiprocessor.

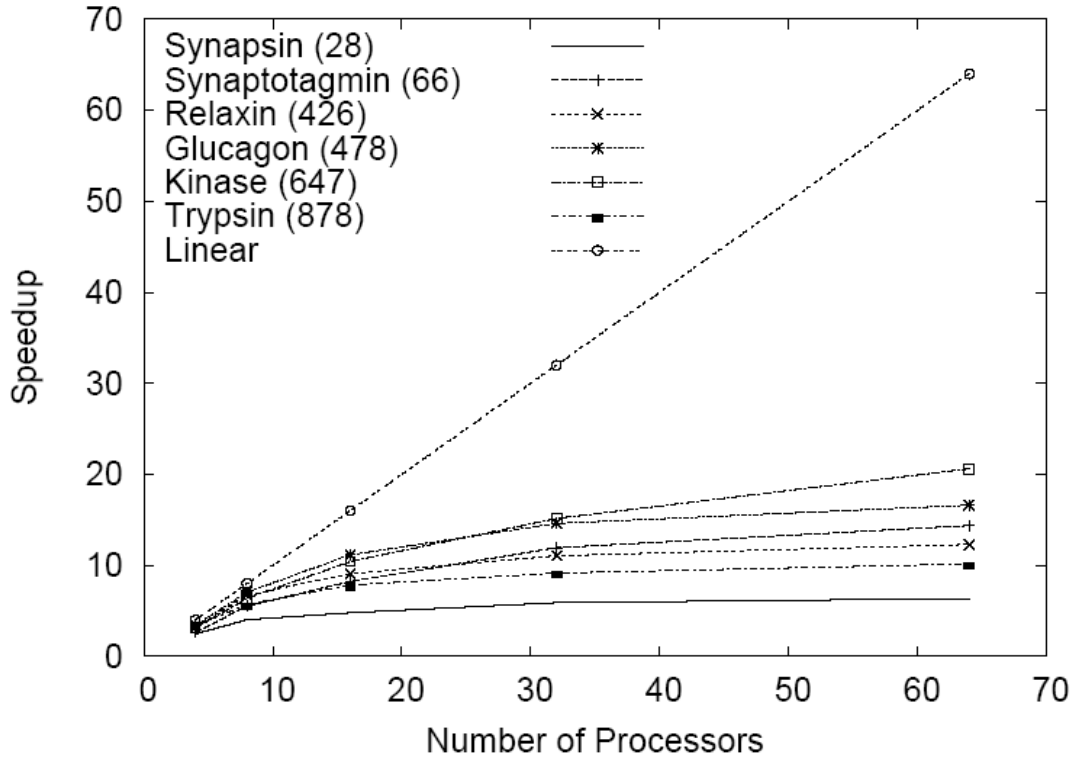

Figure 2.5 Experiment result of pCLUSTAL

The sequences were, on average, about 300 amino acids in length. The speedup is very good for up to 8 processors, good for up to 16 processors, and still reasonable for up to 32 processors.

2.5.3 ClustalW-MPI

The paper cited: ClustalW-MPI: ClustalW analysis using distributed and parallel computing [11]

ClustalW-MPI is a distributed and parallel implementation of ClustalW. All three steps have been parallelized to reduce the execution time. The software uses a message-passing library called MPI (Message Passing Interface) and runs on distributed workstation clusters as well as on traditional parallel computers.

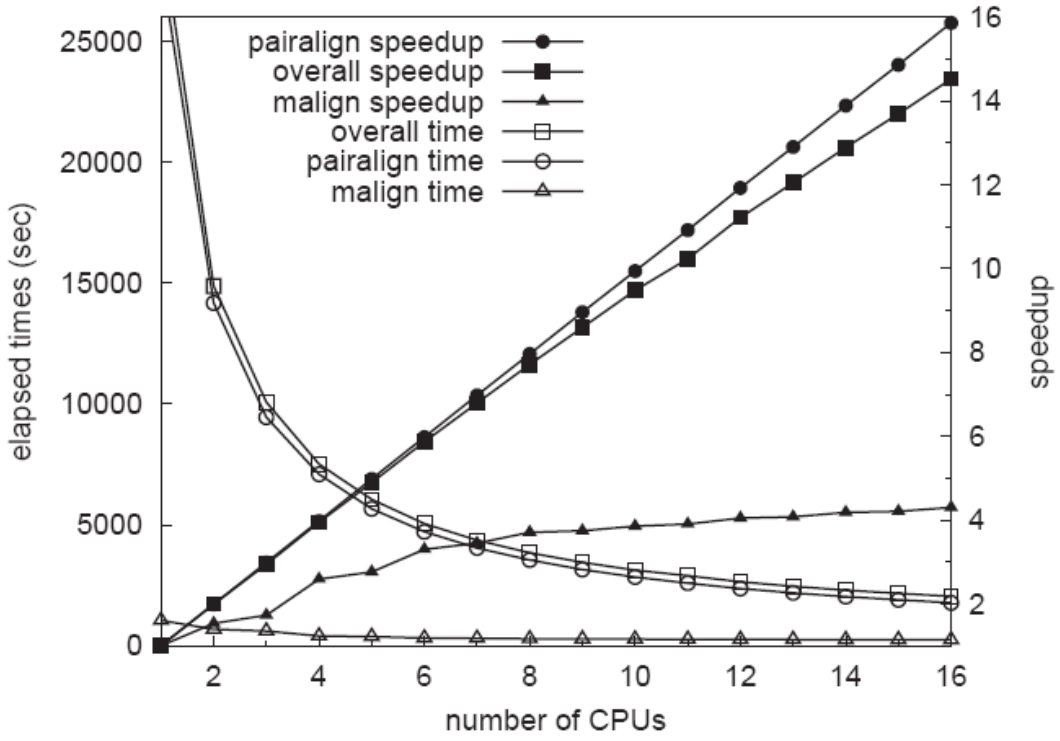

Figure 2.6 Experiment result of ClustalW-MPI

The graph shows elapsed times and speedups for the ClustalW-MPI results of the 500-sequence data with an average length of about 1100 amino acids. Pairalign is

the CPU time for the calculation of pairwise distance, malign is the CPU time for progressive alignment.

2.5.4 Parallel Multiple Sequence Alignment

The paper cited: Parallel Multiple Sequence Alignment with Dynamic Scheduling [12]

This paper proposes a parallel implementation of the multiple sequence alignment algorithms, known as ClustalW, on distributed memory parallel machines. The proposed algorithm divides a progressive alignment into subtasks and schedules them dynamically. A task tree is built according to the dependency of the generated phylogenetic tree. With dynamic scheduling, tasks are allocated to the processors considering the tasks' estimated computation and communication costs and the processors' workload in order to minimize the completion time. The experiment results show that the proposed parallel implementation is achieves a considerable speedup over the sequential ClustalW.

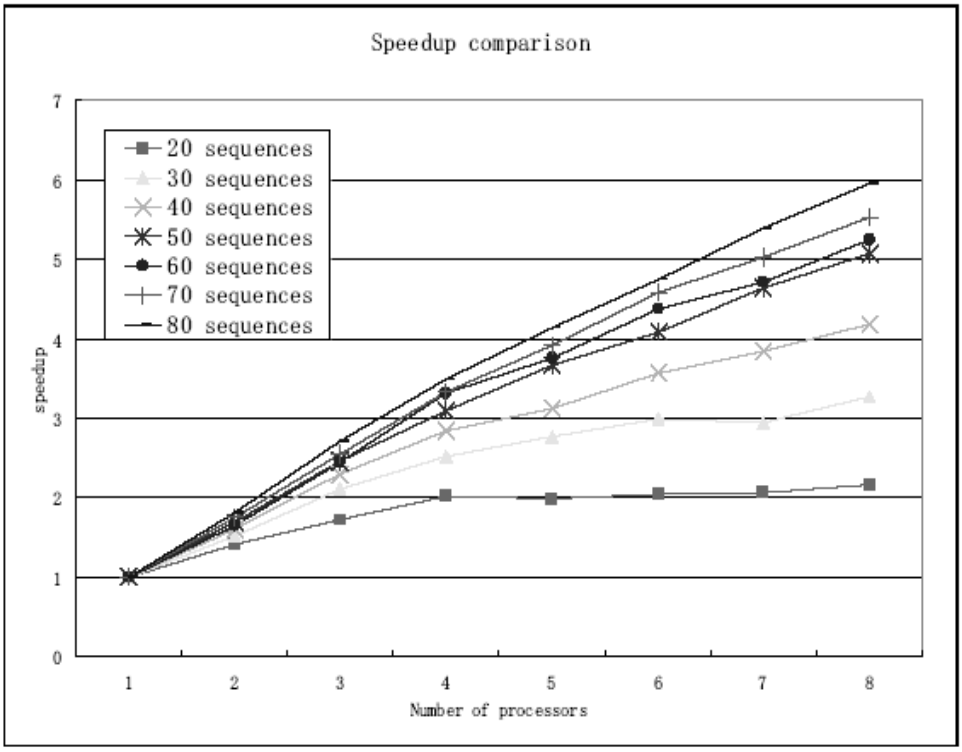

Figure 2.7 Experiment result of Parallel Multiple Sequence Alignment

Bacterial RNA sequences with lengths varying from 289 to 399 nucleotides are used in the experiments. Higher speedup can be achieved when the number of sequences to be aligned is larger. When the number of sequences is large the speedup is close to linear. When the number of sequences is small the speedup will be saturated because the number of tasks on the tree is small.

Chapter 3

Theoretical Backgrounds and Proposed Algorithms

3.1 Amdahl's Law

The theorem that has come to be known as Amdahl's Law was first described by Gene Amdahl in his 1967 paper titled "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". The paper's key statement was:

"If F is the fraction of a calculation that is sequential, and (1-F) is the fraction that can be parallelised, then the maximum speed-up that can be achieved by using P processors is 1/(F+(1-F)/P)."

The point that Amdahl was trying to make was that using lots of parallel processors was not a viable way of achieving the sort of speed-ups that people were looking for. I.e. it was essentially an argument in support of investing effort in making single processor systems run faster. There are at least two somewhat colloquial ways of expressing the essence of Amdahl's Law but in a more general context:

"The performance of any system is constrained by the speed or capacity of the slowest point."

For example, if the "system" is assembling cars and the "system" is able to build one car body every ten minutes and four wheels every five minutes then speeding up the production of wheels won't make any difference to how fast the "system" is able to produce completed cars (assuming that each car requires four wheels). The point of constraint of a system is often referred to as the system's bottleneck. And:

"The impact of an effort to improve the performance of a program is primarily constrained by the amount of time that the program spends in parts of the program not targeted by the effort."

Amdahl's Law is a statement of the maximum theoretical speedup you can ever hope to achieve. The actual speed-ups are always less than the speed-up predicted by

Amdahl's Law. The reasons why the actual speed-up is always less than the theoretical speed-up are many including:

A. Distributing work to the parallel processors and collecting the results back together is extra work required in the parallel version which isn't required in the serial version (i.e. the parallel version has to do more computation than the serial version which results in an increased execution time for the actual parallel version over the theoretical speed of the parallelized serial version).

B. In order to reduce the amount of time spent communicating with other processors, the program is often modified so that certain computations which were performed once in the serial code are performed once by each of the parallel processors. The end result is that these computations are, for all practical purposes, running at the speed of a single processor (i.e. these computations represent parts of the program which havn't been parallelized).

C. Even when the program is executing in the parallel parts of the code, it is unlikely that all of the processors will be computing all of the time as some of them will likely run out of work to do before others are finished their part of the parallel work. This is called the straggler problem because some of the processors are straggling behind when others have finished their work.

Finally, anyone intending to write parallel software simply must have a very deep if not intuitive understanding of Amdahl's Law if they are to avoid having unrealistic expectations of what parallelizing a program/algorithm can achieve and if they are to avoid underestimating the effort required to achieve their performance expectations.

3.2 The Proposed Multithreading Strategy

To change the sequential program to be the multithreading program, we did as follows: 1) Profiling and analysis 2) Applying the thread library and 3) Verification. In this section, we describe our strategy by the case study; ClustalW software.

3.2.1 Profiling and Analysis

First of all, we have to know what the program does. As for ClustalW, we know that it's the multiple sequence alignment software and consists of 3 phases. Now we discuss the complexity for all stages. Give *N* sequences and sequence length *L*, calculating the distance matrix in stage 1 takes $O(N^2L^2)$ time. A neighbor joining algorithm is $O(N^4)$ time for constructing the guide tree in stage 2. And for the last stage, progressive alignment is $O(N^3 + NL^2)$ time. The summary is shown in below table [17].

Table 3.1 Complexity of the sequential ClustalW

Stage	Time complexity
Distance Matrix	$O(N^2L^2)$
Neighbor joining	$O(N^4)$
Progressive alignment	$O(N^3 + NL^2)$
Total	$O(N^4 + L^2)$

Note: The big-O asymptotic complexity of the elements of ClustalW as a function of L, the sequence length, and N, the number of sequences, retaining the highest-order terms in N with L fixed and vice versa.

We have to analyze the program to observe which part is sequential. For our case study, we have analyzed ClustalW-SMP (version 0.99–9); the SMP version of ClustalW version 1.82, was created by O. Duzlevski [18]; to find the bottleneck for improving the throughput and the speedup of the SMP version. We used the Intel thread profiler tool [19] to analyze the SMP version. The profiler shows that the SMP version is not yet fully parallelized in figure 3.1. This figure shows that only Distance matrix and Progressive alignment stages are forced into the threads, but not in Neighbor joining stage. Therefore, we modified the code of both the sequential ClustalW and the SMP versions and measure the execution time of three major stages. The two criteria to be analyzed are: first, the elapsed time of the three stages which we want to find the time

ratio among the three stages and second, the number of threads which we want to find the relation between the thread number and the PC efficiency. The protein sequence data sets from the National Center for Biotechnology Information (NCBI) are used as our test data whose sequence lengths and sequence numbers are shown in table 3.2. The profiles are done on 3GHz Intel Pentium IV processor with hyper thread technology and 1 GB of memory. The elapsed times of ClustalW and ClustalW-SMP are compared which in this case ClustalW-SMP is faster. Although the executions are done in 2, 4 and 8 threads, the execution times of ClustalW-SMP still nearly, that shown in Table 3.3. Table 3.4 exhibits the elapsed times of each stage. In neighbor joining stage, the execution times are similar considering the same sequence numbers while the sequence lengths are fixed. Figure 3.2 and figure 3.3 shows the time ratios of the three stages of ClustalW-SMP with varying the number of sequences and the sequence lengths are fixed at 200 and 800 amino acids. The time ratios of neighbor joining stage with the fewer sequence numbers are more than the time ratios of neighbor joining stage with the more sequence numbers while the sequence lengths are fixed. That means if we can parallel the tasks in the neighbor joining stage, the execute time should be reduced more. This helps the alignment which needs to align the too many sequences to spend the faster time.

Table 3.2 Sequence length and Sequence number of the protein sequence as test data

Variables	Values
Average sequence lengths	200, 800
Sequence number	200, 400, 600, 800
Maximum thread number	2, 4, 8

Table 3.3 Elapsed times of ClustalW compared with ClustalW-SMP running in 2,4,8 threads

#seq-#len	ClustalW(*sec*)	ClustalW-SMP (*sec*)		
		2 threads	4 threads	8 threads
200-200	26	22	22	22
200-800	494	420	414	415

Table 3.3 Elapsed times of ClustalW compared with ClustalW-SMP running in 2,4,8 threads (con.)

#seq-#len	ClustalW(*sec*)	ClustalW-SMP (*sec*)		
		2 threads	4 threads	8 threads
400-200	130	118	116	116
400-800	1,932	1,667	1,665	1,676
600-200	496	469	462	462
600-800	4,544	3,943	3,939	3,959
800-200	1,395	1,347	1,341	1,341
800-800	8,539	7,500	7,480	7,520

Table 3.4 Elapsed times of ClustalW-SMP in each stage

#seq-#len	ClustalW-SMP (*sec*)		
	Distance Matrix	Neighbor Joining	Progressive Alignment
200-200	17	1	3
200-800	379	1	34
400-200	65	41	10
400-800	1,538	41	86
600-200	149	293	21
600-800	3,481	293	166
800-200	259	1,044	38
800-800	6,185	1,050	265

Figure 3.1 Intel Thread Profiler shows the time-line of the experiment with 400 protein sequences and about 200 amino acids in length.

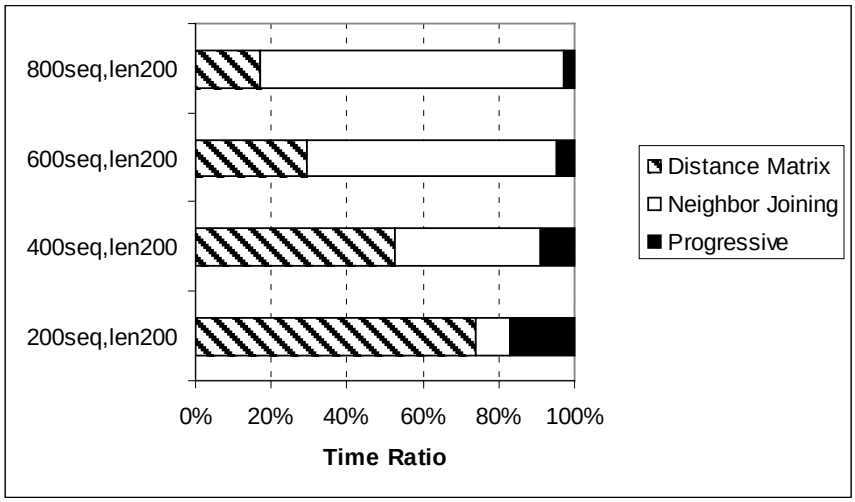

Figure 3.2 Time ratio of the three stages of ClustalW-SMP. The numbers of sequences are varied but the sequences lengths are fixed at about 200 amino acids

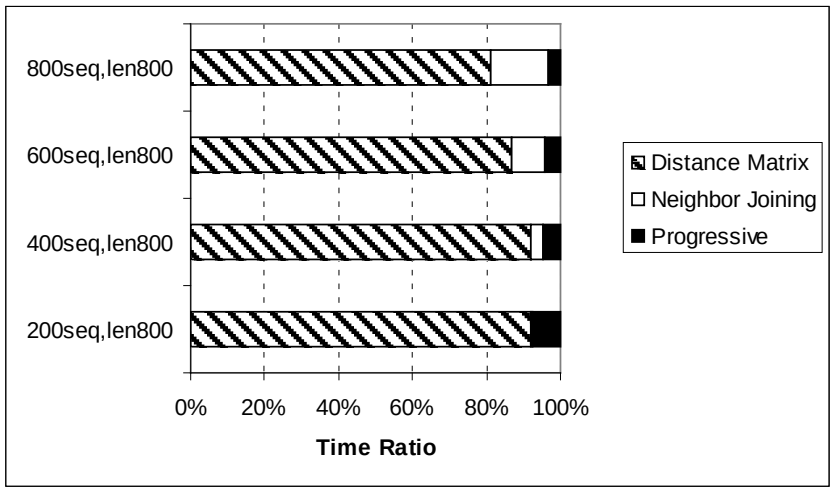

Figure 3.3 Time ratio of the three stages of ClustalW-SMP. The numbers of sequences are varied but the sequences lengths are fixed at about 800 amino acids

3.2.2 Applying the Thread Library

After profiling, we have modified the target part which is the sequential part. As for our case study, we have modified the neighbor joining part of ClustalW-SMP. The implementation was done by using pthread library with MUTEX object as a synchronization object. Figure 3.4 shows the code for guide tree stage, the 4 nested loops has $O(N^4)$ time complexity. We then break the 2 inner loops to be executed in parallel. The flowchart of the thread synchronization is described in figure 3.5. After the 2 inner loops were moved to be the parallel function, the parameters are set in the main thread. The main thread starts the parallel function by calling *signal(start)* and waits for the reply signal to execute next loop. The thread num variable is the maximum number of thread that we can use. In parallel function, it executes parallel tasks after waiting for the start signal. Then the parallel function decreases the thread num variable by one and sends the thread num signal to the main thread. After calling *signal(thread num)*, the parallel function reaches the while condition and determine if there are more tasks to be executed. If so, the parallel function continues waiting for the start signal but if not, the function will cease. Back to the main thread, the main thread proceeds to the next loop after all parallel threads complete. When all loops are done, the main thread continues in another part. We reduce pthread creation overhead by creating the array of parallel functions. The set is equal to the maximum thread number that is set the user. All loads are distributed to all thread functions equally and taken by the parallel functions to execute the tasks. The logical diagram of the algorithm shows in figure 3.6.

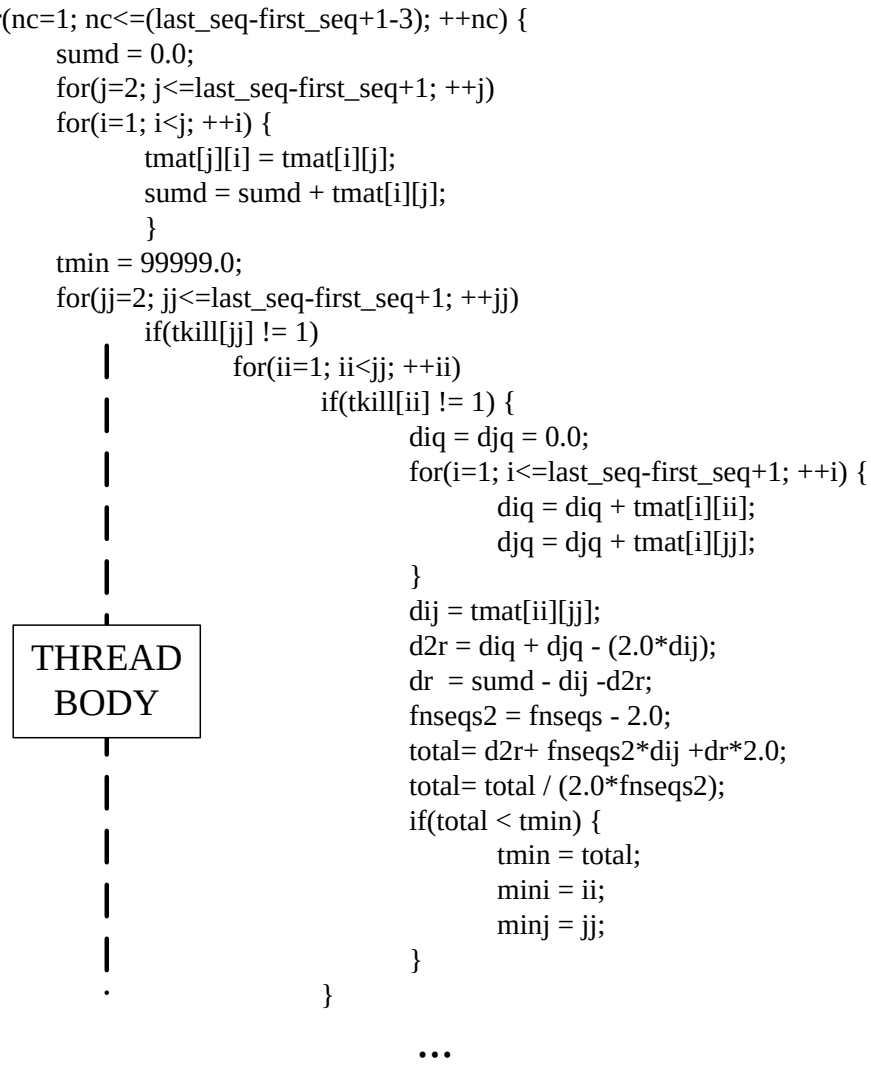

Figure 3.4 The source code of the guide tree stage

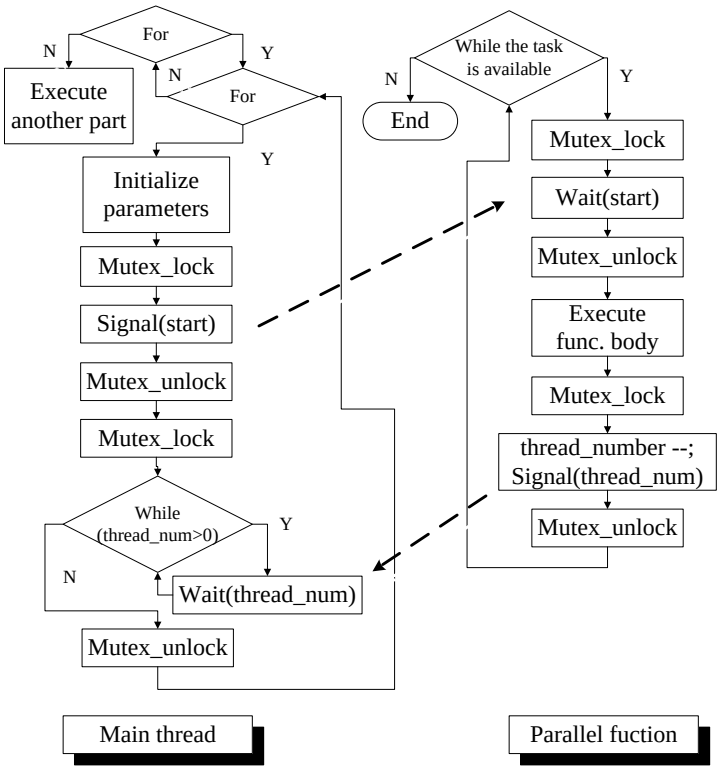

Figure 3.5 Flowchart of the thread synchronization

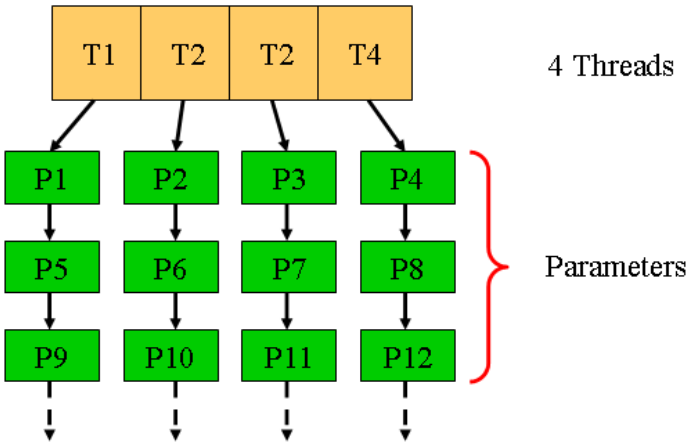

Figure 3.6 All loads are distributed to all thread functions equally as the parameters

3.2.3 Verification

After we have completed in modifying the program, we have to verify for the correctness of the result and check how much speedup that the modified program performs. We use Beyond Compare [22] to verify the result as figure 3.7.

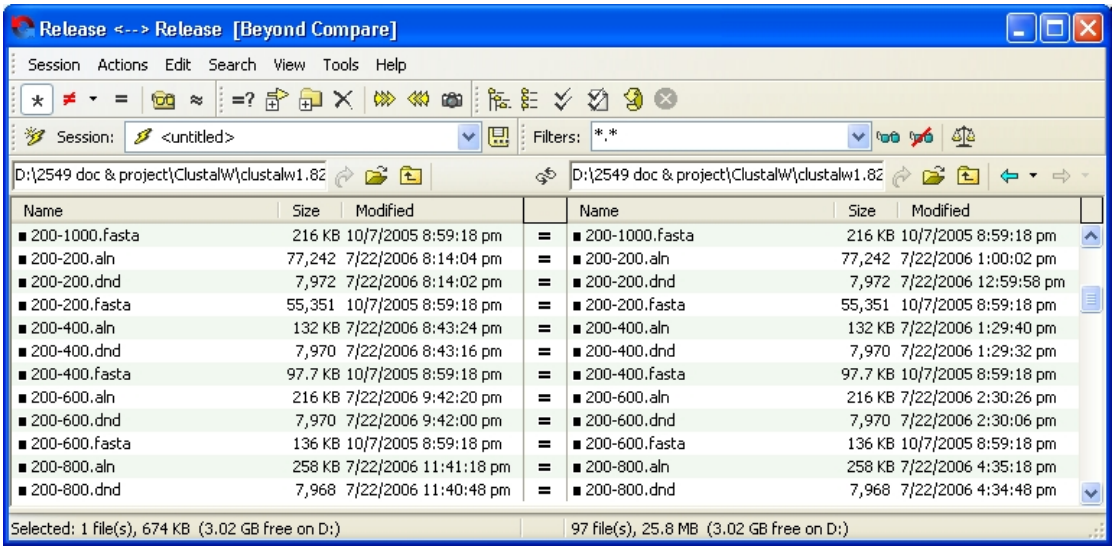

Figure 3.7 Verify the result using Beyond Compare

3.3 Proposed Optimizing and Vectorizing Methodology

Normally, we can optimize the program by setting the optimizing. More advance, we can optimize the program to vectorize the loop for faster execution. In our work, we did as follows: 1) Profiling 2) Applying the loop optimizing methodologies and 3) Verification. In this section, we describe our methodology by the case study; ClustalW software.

3.3.1 Profiling

Use the profiling tool to find the top usage functions, in our experiment; we used the Intel C++ Compiler for Windows to optimize the MT-ClustalW by enabling the /QxP option which optimizes code for Intel Core Duo processors and Intel Core Solo processors, Intel Pentium 4 processors with Streaming SIMD Extensions 3, and compatible Intel processors with Streaming SIMD Extensions 3 (SSE3). With the /QxP option, the compiler automatically traces the code and optimizes the loops. As we have done, only the simple loops were optimized. So we had to profile and change some codes to assist the compiler for optimizing the loops. We used the Intel VTune Performance Analyzer to profile ClustalW in debug mode and look for the hotspots within the functions. The profiling result is shown in the below table.

Table 3.5 Profiling result of ClustalW (debug) from the Intel VTune Performance Analyzer

Function	Clockticks (%)
diff	33.36
prfscore	15.93
forward_pass	14.91
calc_score	12.93
reverse_pass	11.45
pdiff	5.85

3.3.2 Applying the Loop Optimizing Methodologies

After profiling, we focus on the top-usage functions to search for the hotspots and modify the codes with our guide methodologies. Here are the methodologies that we used and applied to the codes for assisting the compiler to optimize the code better than original ClustalW 1.8. The methodologies are described as follows: Loop reversal, Loop fission, Type Casting, and Reduce procedure calls. The profiling result and the applied methodologies were shown in below table.

Table 3.6 Applied optimization methodology

Function	Optimization methodology*
diff	A,B
prfscore	C
forward_pass	-
calc_score	D
reverse_pass	A
pdiff	-

*Note: A is Loop reversal, B is Loop fission, C is Type Casting, and D is Procedure call reduction

A. Loop reversal

That is to run a loop backward. Reversal of for loops is always legal, since the execution is not defined in terms of the order of the index set. Thus, loop reversal is legal only when the loop carries no dependence relations. Here the example code, this loop is not vectorized by the compiler.

```
for (i=se2;i>0;i--)
{
        HH[i] = -1;
        DD[i] = -1;
}
```

After we reverse the loop, it can be vectorized as follows.

```
for (i=1;i<=se2;i++)
{
        HH[i] = -1;
        DD[i] = -1;
}
```

B. Loop fission

A single loop can be broken into two or more smaller loops. Loop fission can break up the block of conditionally executed statements. To apply loop fission, a temporary array must be used to hold the result of the variable which is not array. The temporary array is used to guard the execution of the statements after fission. The example code is shown as follow:

```
for (j=0;j<=N;j++)
{
        hh = HH[j] + RR[j];
        if (hh>=midh)
        if (HH[j]!=DD[j]&&RR[j]==SS[j])
        {
                midh=hh;
                midj=j;
        }
}
```

After applying loop fission, the code contains 2 loops. The first loop can be vectorized, but the second loop can not. However, we can make the compiler vectorize the second loop by using a temporary array to hold the result of the conditional test.

```
for (j=0;j<=N;j++)
{
        temp[j] = HH[j] + RR[j];
}
for (j=0;j<=N;j++)
{
        if (temp[j]>=midh)
        if (HH[j]!=DD[j]&&RR[j]==SS[j])
        {
                midh=temp[j];
                midj=j;
        }
}
```

C. Type Casting

That is converting an expression of a given type into another type. Type casting can be used to promote the variable from integer to float, this can help the compiler to vectorize the loop using the Intel MMX technology. The example is shown as follow:

```
for (ix=0;ix<=max_aa;ix++)
{
        score+=(profile1[n][ix]*profile2[m][ix]);
}
```

After applying this method, the loop is vectorized. However, don't forget to convert the result of the statement to the original type.

```
for (ix=0;ix<=max_aa;ix++)
{
        score+=(int)((float)profile1[n][ix]* (float)profile2[m][ix]);
}
```

D. Procedure call reduction

Calling the procedure in the loop, the processor takes much overhead. The code can reduce the procedure calls by using Macro. The example code can be rewrite as Macro and is shown as follow:

```
static sint calc_score(sint iat,sint jat,sint v1,sint v2)
{
        sint ipos,jpos,ret;
        ipos = v1 + iat;
        jpos = v2 + jat;
        ret=matrix[(int)seq_array[seq1][ipos]][(int)seq_array[seq2][jpos]];
        return(ret);
}
```

After rewriting as Macro, the execution time greatly reduces because there isn't need to process the procedure call during the nested loops and the recursions. The Macro code is like this:

```
#define calc_score(iat,jat,v1,v2)
        matrix[(int)seq_array[seq1][v1+iat]][(int)seq_array[seq2][v2 + jat]]
```

3.3.3 Compiling Result and Verification

After modifying the program, check the compiling result if the loops are vectorized or not. Then run and verify the alignment result using Beyond Compare. Finally, check how much speedup that the optimized program performs.

```
\showpair.c(51) : (col. 2) remark: LOOP WAS VECTORIZED.
\showpair.c(55) : (col. 2) remark: LOOP WAS VECTORIZED.
\showpair.c(57) : (col. 2) remark: LOOP WAS VECTORIZED.
\showpair.c(89) : (col. 2) remark: LOOP WAS VECTORIZED.
\showpair.c(91) : (col. 2) remark: LOOP WAS VECTORIZED.
\showpair.c(249) : (col. 3) remark: LOOP WAS VECTORIZED.
\showpair.c(244) : (col. 3) remark: LOOP WAS VECTORIZED.
\showpair.c(292) : (col. 4) remark: LOOP WAS VECTORIZED.

\trees.c(943) : (col. 4) remark: PARTIAL LOOP WAS VECTORIZED.
\trees.c(943) : (col. 4) remark: PARTIAL LOOP WAS VECTORIZED.
\trees.c(480) : (col. 3) remark: LOOP WAS VECTORIZED.
\trees.c(563) : (col. 2) remark: LOOP WAS VECTORIZED.
```

Figure 3.8 Compiling result

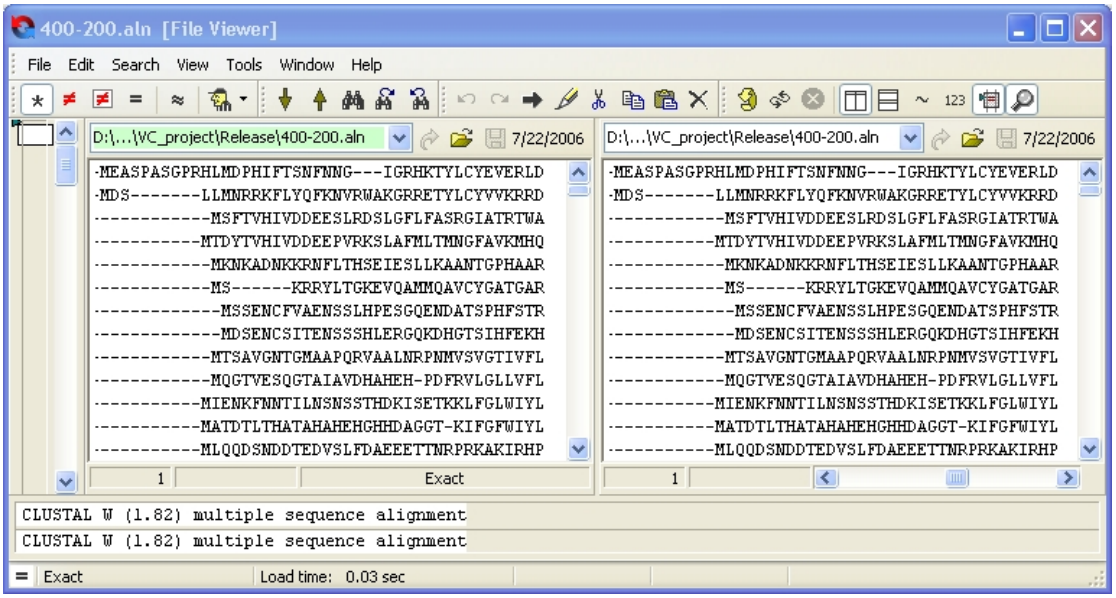

Figure 3.9 Verify the alignment result using Beyond Compare

Chapter 4

Experiment Results and Discussions

4.1 Multithreading Implementation Experiments and Results

We implement our algorithm in C programming language and utilize the pthread library. The experiment was conducted using a 2.8GHz Intel Pentium D (Dual core) processor with 2 GB of memory. This computer runs MS Windows XP pro service pack 2 with no other applications installed. The elapsed times are measured as well as the wall clock time between the distance matrix stage and the progressive alignment stage. All measured times include the times that were taken to read the input data from a file and write the solution into a file. Furthermore, all measured times were measured when there is no one using the system except this experiment. Our experiments measured the following: (1) The elapsed times as a function of the number of sequences in the guide tree stage. (2) Relative speedup (ratio of the elapsed time of ClustalW and the elapsed time of MT-ClustalW) as a function of the number of sequences. The test data and experiment parameters are shown in the table 4.1. With the same test data set, figure 4.1 shows the elapsed times for the ClustalW and MT-ClustalW results of the Neighbor joining stage as a function of number of sequences when running 8 threads. The MT-ClustalW successfully decreases the execution time. Figure 4.2 shows the speedup for the ClustalW-SMP and MT-ClustalW results of 8 threads as a function of number of sequences as compared to ClustalW. It can be observed that higher speedup can be achieved with the shorter sequence length to be aligned. When the sequence length is small, the speedup increases. When the sequence length is large, the speedup will decrease because the elapsed time of the guide tree stage depend on the number of sequences but the other stages depend on both the number of sequences and the sequence length. If the number of sequences is large and the sequence length is long, the ratio of the elapsed time of the guide tree stage and the elapsed time of the other stages is lower so the speedup decrease and vice versa. Now, we will focus on the speedup of MT-ClustalW as a function of number of threads. In figure 4.3, the speedup increases from 2 threads to 4 threads then decreases after 4 threads. Also in figure 4.4,

the speedup is almost the same as the speedup of figure 4.3 except the 800 sequence data. The speedup seems to be lower then steady when the thread number is raised because the limitation of the machine resources. The proposed method fully utilizes the multithreading feature in ClutalW and achieves the better speedup of ClustalW. MT-ClustalW is faster than ClustalW-SMP especially when the number of sequences is large. This will be compatible with the massive work for the alignment and gains more throughput.

Table 4.1 Parameters in the experiment

Parameters	Values
Average sequence lengths	200, 400, 600, 800
Sequence number	200, 400, 600, 800
Maximum thread number	2, 4, 8
Number of experiments	3 times

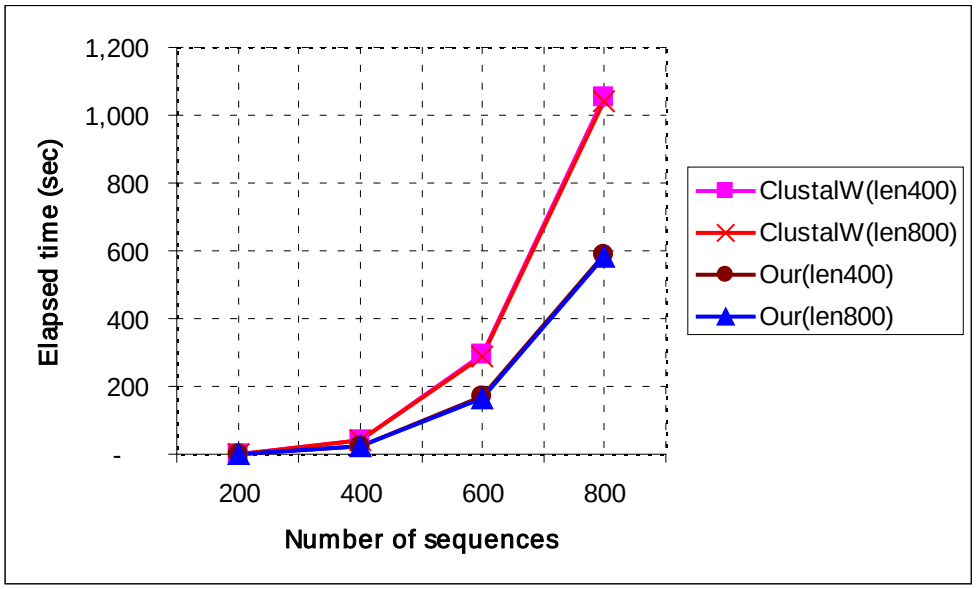

Figure 4.1 Elapsed times for the ClustalW and MT-ClustalW results of Neighbor joining stage as a function of number of sequences (8 threads)

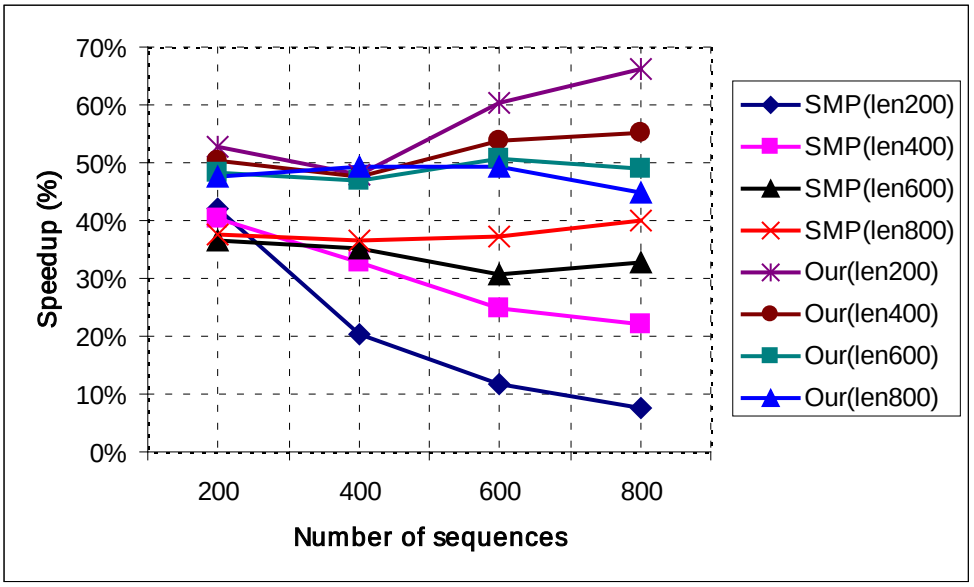

Figure 4.2 Speedup for the ClustalW-SMP and MT-ClustalW results of 8 threads as a function of number of sequences. Both speedup are compared with ClustalW

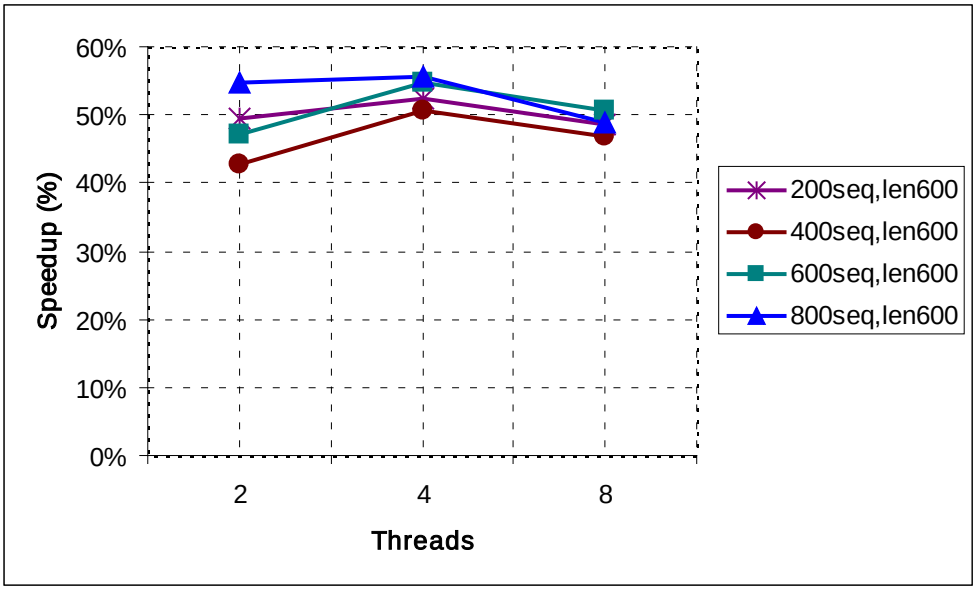

Figure 4.3 Speedup of the MT-ClustalW results as a function of number of threads that compared to ClustalW. The sequence lengths are fixed at 600 amino acids

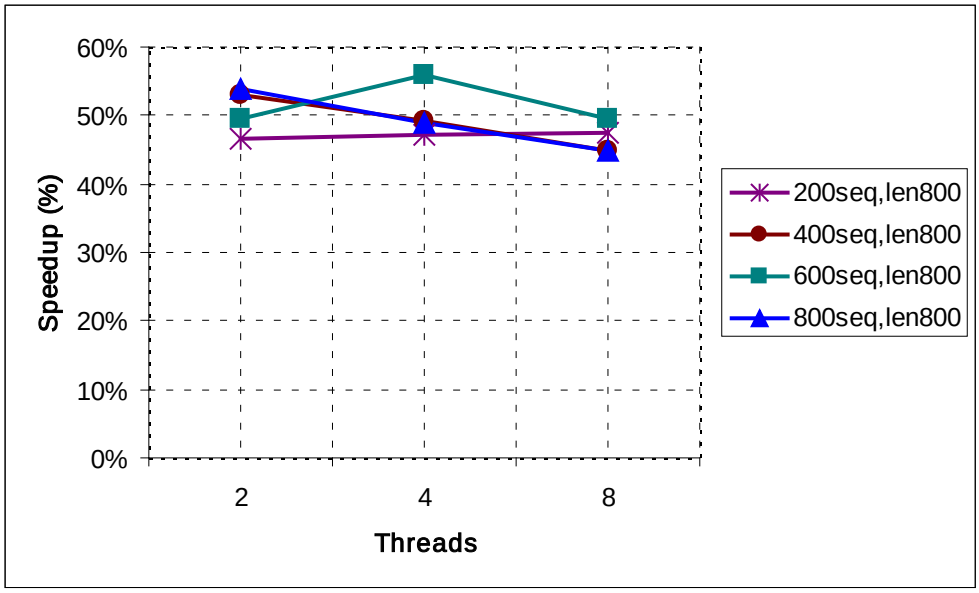

Figure 4.4 Speedup of the MT-ClustalW results as a function of number of threads that compared to ClustalW. The sequence lengths are fixed at 800 amino acids

4.2 Optimizing and Vectorizing Implementation Experiments and Results

We implement our method using Intel C++ Compiler 9.0. The experiment was conducted using a 2.8GHz Intel 820 Pentium D (Dual core) processor with 2 GB of memory. This computer runs MS Windows XP pro service pack 2 with no other applications installed. The elapsed times are measured as well as the wall clock time between the distance matrix stage and the progressive alignment stage. All measured times include the times that were taken to read the input data from a file and write the solution into a file. Furthermore, all measured times were measured when there is no one using the system except this experiment. Our experiments measured as follows: (1) The elapsed times as a function of the number of sequences in every stages. (2) The relative speedup (ratio of the elapsed times of both ClustalW and MT-ClustalW and the elapsed times of our assist version of both ClustalW and MT-ClustalW) as a function of the number of sequences. We have done the whole experiment for 3 times with the test data set as shows in Table 4.2. We ran the MT-ClustalW with 4 threads as refer from our previous work. With the different running mode, Table 4.3 shows the elapsed times of ClustalW and MT-ClustalW in each stage and the overall speedup. The elapsed times mainly reduce in the distance matrix stage for both ClustalW and MT-ClustalW, but little

reduces in the neighbor Joining stage and the progressive Alignment stage. And the results are quite similar though the test data has the different size. With the same experiment, the speedups increase from Running mode II to Running mode VI, in order. All speedups are compared with the sequential ClustalW or Running mode I. With the same test data set, figure 4.5 shows the speedups of the optimized versions of ClustalW as a function of number of sequences. All speedups are compared with original ClustalW. It can be observed that ClustalW which is optimized with our assist gains significantly higher speedup than only compileroptimization. Also when the sequence length changes from 400 to 600 amino acids, ClustalW which is optimized with our assist achieves higher speedup. However, the speedups of ClustalW of both versions decrease when the number of sequences is large. Also similar with figure 4.5, we change the test data to be 800 to 1000 amino acids in figure 4.7. As same as ClustalW result, figure 4.6 shows the speedups of the optimized versions of MT-ClustalW, but MT-ClustalW which is optimized with our assist gains much higher speedup when the sequence length is fixed at 600 amino acids. And also similar with figure 4.6, we change the test data to be 800 to 1000 amino acids in figure 4.8. The proposed method fully utilizes the optimization feature in ClutalW and achieves the better speedup of ClustalW. MTClustalW is faster than ClustalW especially when the number of sequences is large. This will be compatible with the massive work for the alignment and gains more throughput.

Table 4.2 Parameters of the experiment and the test data set

Parameter	Value
Number of experiments	3 times
Number of threads	4 threads
Sequence lengths	200, 400, 600, 800
Number of sequences	800, 1000

Table 4.3 Elapsed times in each stage and overall speedup of ClustalW and MT-ClustalW

Running mode*	Elapsed times (*ms*)			Overall speedup
	Distance Matrix	Neighbor Joining	Progressive Alignment	
Test data #1 - 200 sequences, 800 amino acids				
I	448,922	1,485	30,078	-
II	386,375	1,297	30,468	1.15
III	361,297	1,297	29,188	1.23
IV	264,172	1,234	19,859	1.68
V	220,312	1,063	19,781	1.99
VI	205,297	1,078	18,922	2.13
Test data #2 - 800 sequences, 1000 amino acids				
I	11,918,672	932,718	333,110	-
II	10,387,046	881,125	338,016	1.14
III	9,656,750	880,969	327,985	1.21
IV	7,009,875	511,047	252,984	1.70
V	5,900,891	473,359	253,188	1.98
VI	5,472,407	474,109	244,672	2.12

*Note: Running mode defines as follows: (I) ClustalW without optimization (II) ClustalW with optimization (III) ClustalW with optimization and our assist (IV) MT-ClustalW without optimization (V) MT-ClustalW with optimization (VI) MT-ClustalW with optimization and our assist

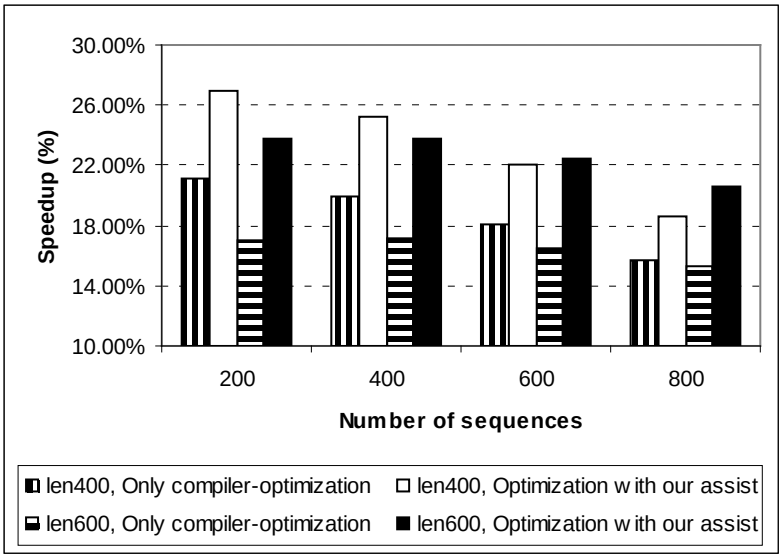

Figure 4.5 Speedup of the optimized versions of ClustalW as a function of number of sequences. The sequence lengths are fixed at 400 and 600 amino acids. All speedups are compared with the original sequential ClustalW

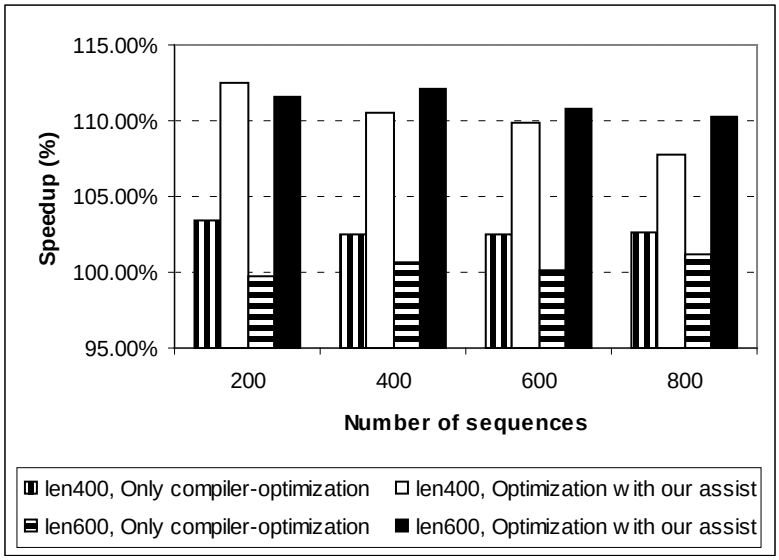

Figure 4.6 Speedup of the optimized versions of MT-ClustalW as a function of number of sequences. The sequence lengths are fixed at 400 and 600 amino acids. All speedups are compared with the original sequential ClustalW

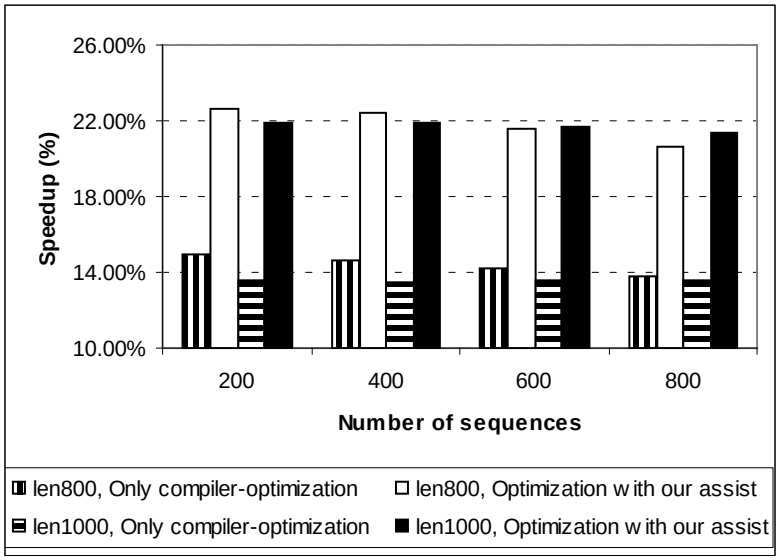

Figure 4.7 Speedup of the optimized versions of ClustalW as a function of number of sequences. The sequence lengths are fixed at 800 and 1000 amino acids. All speedups are compared with the original sequential ClustalW

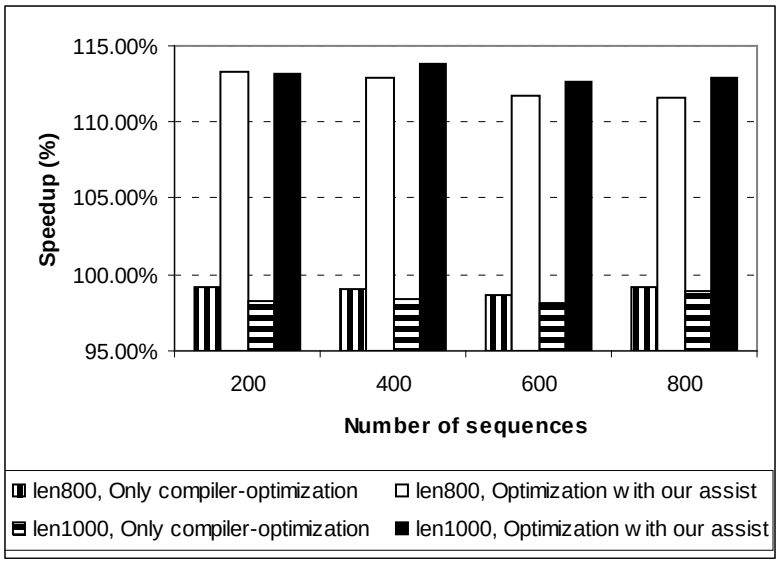

Figure 4.8 Speedup of the optimized versions of MT-ClustalW as a function of number of sequences. The sequence lengths are fixed at 800 and 1000 amino acids. All speedups are compared with the original sequential ClustalW

4.3 Overall Results and Discussions

This section, we describe the result with Amdahl's Law. In essence, Amdahl's Law states that the overall speedup of a computer system depends on both the speedup in a particular component and how much that component is used by the system. In symbols:

$$S = \frac{1}{(1-f)+\frac{f}{k}} \quad (1)$$

where *S* is the overall system speedup, *f* is the fraction of work performed by the improved part, and *k* is the speedup of a new component. For the vectorized optimization, we can approximate that k is 4 because the processor can execute 4 data for each instruction. As referred to (1), we can calculate f as:

$$f_{opt} = -\frac{4}{3}\left[\frac{1-S}{S}\right] \quad (2)$$

where f_{opt} is the improved fraction of the optimized version of ClustalW. For the experiment of ClustalW with fully optimization testing with 800 sequences (length is 1000 amino acids), the total speedup is 1.21 times as in table III, therefore we can find the improved fraction of ClustalW with optimization as in (2), we have:

$$f_{opt} = -\frac{4}{3}\left[\frac{1-1.21}{1.21}\right] = 0.23$$

So we can conclude that the fraction of the improved part is 23% for this experiment.
And for the fully multithread version of ClustalW running on the dual core processors, we can approximate that k is 2. As referred to (1) , we can calculate f as:

$$f_{mt} = -2\left[\frac{1-S}{S}\right] \quad (3)$$

where f_{mt} is the improved fraction of the multithreading ClustalW. With the same test data, we can find the improved fraction of the multithreading ClustalW as in (3), so we have:

$$f_{mt} = -2\left[\frac{1-1.70}{1.70}\right] = 0.82$$

We can conclude that the fraction of the improved part is 82% for this experiment. The multithreading fraction is bigger than the optimizing fraction, so that applying multithreading algorithm can gain higher speedup. Next, we discuss about the overall speedup, the fully multithread version of ClustalW with optimization should gain as follow:

$$Speedup_{total} = Speedup_{opt} \times Speedup_{mt} \qquad (4)$$

where $Speedup_{total}$ is the overall system speedup, $Speedup_{opt}$ is the speedup of ClustalW with optimization, and $Speedup_{mt}$ is the speedup of the multithreading ClustalW. As the example experiment, we can calculate the total speedup as in (4), so we get:

$$Speedup_{total} = 1.21 \times 1.70 = 2.06$$

As referred to the experiment result, the completed version of ClustalW achieve 2.13 times faster that relates to the above discussion.

Next, we discuss about the complexity of the optimized multithreading ClustalW. Suppose the application which was optimized with our assist runs on the machine that contains k processors. Give N sequences and sequence length L, calculating the distance matrix in stage 1 takes $\frac{N^2L^2}{4k} \rightarrow O\left(N^2L^2\right)$ time. A neighbor joining algorithm is $\frac{N^4}{4k} \rightarrow O\left(N^4\right)$ time for constructing the guide tree in stage 2. And for the last stage,

progressive alignment is $\frac{N^3 + NL^2}{4k} \rightarrow O\left(N^3 + NL^2\right)$ time. So the total is $O\left(N^4 + L^2\right)$ time. The summary is shown in below table.

Table 4.4 Complexity of the optimized multithreading ClustalW

Stage	Original	Ex. *k* processors, vertorized	Final
Distance Matrix	$O\left(N^2L^2\right)$	$\frac{N^2L^2}{4k}$	$O\left(N^2L^2\right)$
Neighbor joining	$O\left(N^4\right)$	$\frac{N^4}{4k}$	$O\left(N^4\right)$
Progressive alignment	$O\left(N^3 + NL^2\right)$	$\frac{N^3 + NL^2}{4k}$	$O\left(N^3 + NL^2\right)$
Total	$O\left(N^4 + L^2\right)$	➔	$O\left(N^4 + L^2\right)$

Finally, we compare the optimized MT-ClustalW with the other related works. We compare in terms of sequence numbers, sequence lengths, machince, number of processers, and speedup. We can't do the experiment for all works, so we compare all results at the nearly parameters. We choose the number of sequnces at 500-647 and the sequence length at 300-400. We compare the results of both single machine and PC cluster. We get that our optimized multithreading ClustalW gain higher speedup than the others using 2 processors and over 2 times faster. The comparison summary is shown in the below table.

Table 4.5 Comparison of the optimized MT-ClustalW and the other related works

	pClustal [10]	SGI [9]	ClustalW-MPI [11]	Parallel MSA [12]	ClustalW-SMP	MT-ClustalW	Optimized MT-ClustalW
Number of sequences	647	600	500	80	600	600	600
Sequence length	300	390	1100	289-399	400	400	400
Machine	PC Cluster	Single PC Shared memory	PC Cluster	PC Cluster	Single PC Shared memory	Single PC Shared memory	Single PC Shared memory
Processors	4	2	2	2	2	2	2
Speedup	3.5x	1.8x	1.85x	1.8x	1.25x	1.52x	2.25x

Chapter 5

Conclusions

In this thesis, we presented the multithreading strategies and the optimizing and vectorizing methodologies to improve a sequential application. We demonstrate and analyze one of Bioinformatics software, ClustalW, for our case study. As for our achieved work, we presented a fully multithreading version of ClustalW called MT-ClustalW which significantly improves the performance of the traditional ClustalW. In the proposed MT-ClustalW, all stages: the distance matrix, the guide tree and the progressive alignment; are divided into a number of threads and executed on a Pentium-D machine. The proposed strategy shows better speedups than the ClustalW-SMP and better than that of sequential ClustalW.

In addition, we presented an optimized version of MTClustalW which is done with our proposed methodologies that are able to assist the compiler in optimizing. The optimized MT-ClustalW is also done for a Pentium-D machine by using the Intel C++ compiler. The proposed methodologies achieve better speedups than the original MT-ClustalW.

Bibliography

[1] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu, “A tool for multiple sequence alignment,” *Proc. Nail. Acad. Sci. USA*, vol. 86, pp. 4412–4415, 1989.

[2] V. A. Simossis and J. Heringa, “Praline: a multiple sequence alignment toolbox that integrates homology-extended and secondary structure information,” *Nucleic Acids Research, 2005*, vol. 33, pp. 289–294, 2005.

[3] J. D. Thompson, T. J. Gibson, F. Plewniak, F. Jeanmougin, and D. G. Higgins, “The clustal x windows interface: flexible strategies for multiple sequence alignment aided by quality analysis tools,” *Nucleic Acids Research, 1997*, vol. 25, no. 24, pp. 48764882, 1997.

[4] T. Rognes and E. Seeberg, “Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors,” *Bioinformatics 2000*, vol. 16, no. 8, pp. 699–706, 2000.

[5] X. Meng and V. Chaudhary, “Bio-sequence analysis with cradles 3soc software scalable system on chip,” *ACM Symposium on Applied Computing*, 2004.

[6] J.A. Grice, R. Hughey, and D. Speck, “Parallel sequence alignment in limited space,” *Proceedings of International Conference Intelligent Systems for Molecular*, pp. 145–153, 1995.

[7] F. Lau, “An integrated approach to fast, sensitive, and cost-effective smith-waterman multiple sequence alignment,” *Bioinformatics Module*, 2000.

[8] http://www.sgi.com, “Silicon graphics, inc,” .

[9] D. Mikhailov, Haruna C., and R. Gomperts, “Performance optimization of clustal w: Parallel clustal w, ht clustal, and multiclustal,” *SGI ChemBio*.

[10] J. Cheetham, F. Dehne, S. Pitre, A. Rau-Chaplin, and P. J. Taillon, “Parallel clustal w for pc clusters,” *Proceedings of International Conference on Computational Science and Its Applications (ICCSA)*, vol. 2668, pp. 300–309, 2003.

[11] K.B. Li, “Clustalw-mpi: Clustalw analysis using distributed and parallel computing,” *Bioinformatics 19*, vol. 19, no. 12, pp. 1585–1586, 2003.

[12] J. Luo, I. Ahmad, M. Ahmed, and R. Paul, “Parallel multiple sequence alignment with dynamic scheduling,” *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC’05)*, 2005.

[13] D. Nathan, R. Clemens, T. Oliver, B. Schmidt, and D. Maskell, “Multiple sequence alignment on an fpga,” *Proceedings of the Forth IEEE Int. Workshop on High Performance Computational Biology (HiCOMB 2005, IPDPS 2005)*, 2005.

[14] U. Catalyurek, E. Stahlberg, R. Ferreira, and J. Saltz, “Improving performance of multiple sequence alignment analysis in multi-client environments,” *Proceedings of the First IEEE Int. Workshop on High Performance Computational Biology (HiCOMB 2002, IPDPS 2002)*, 2002.

[15] U. Catalyurek, M. Gray, T. Kurc, J. Saltz, E. Stahlberg, and R. Ferreira, “A component-based implementation of multiple sequence alignment,” *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC2003)*, pp. 122–126, 2003.

[16] N. Saitou and M. Nei, “The neighbor-joining method: a new method for reconstructing phylogenetic trees,” *Mol Biol Evol*, vol. 4, no. 4, pp. 406–425, 1987.

[17] R.C. Edgar, “Muscle: a multiple sequence alignment method with reduced time and space complexity,” *BMC Bioinformatics. 2004*, vol. 5, 2004.

[18] http://bioinfo.pbi.nrc.ca, “Clustalw smp ver. 0.99-9,” 2002.

[19] http://www.intel.com, “Intel thread profiler,” .

[20] http://www.intel.com, “Intel c++ compiler for windows,” .

[21] http://www.hayestechnologies.com/en/techsimd.htm, “Technology: SIMD / MMX / SSE / SSE2 / 3DNow!,” .

[22] http://www.scootersoftware.com, “Beyond Compare,” .

Appendix A
Multithreading Supports on Windows

A.1 What is Multithreading?

Before beginning, it is necessary to define precisely what is meant by the term multithreading. Multithreading is a specialized form of multitasking. In general, there are two types of multitasking: process-based and thread-based. A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, it is process-based multitasking that allows you to run a word processor at the same time you are using a spreadsheet or browsing the Internet. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

A thread is a dispatchable unit of executable code. The name comes from the concept of a "thread of execution." In a thread-based multitasking environment, all processes have at least one thread, but they can have more. This means that a single program can perform two or more tasks concurrently. For instance, a text editor can be formatting text at the same time that it is printing, as long as these two actions are being performed by two separate threads. The differences between process-based and thread-based multitasking can be summarized like this: Process-based multitasking handles the concurrent execution of programs. Thread-based multitasking deals with the concurrent execution of pieces of the same program.

In the preceding discussions, it is important to clarify that true concurrent execution is possible only in a multiple-CPU system in which each process or thread has unrestricted access to a CPU. For single CPU systems, which constitute the vast majority of systems in use today, only the appearance of simultaneous execution is achieved. In a single CPU system, each process or thread receives a portion of the CPU's time, with the amount of time determined by several factors, including the priority of the process or thread. Although truly concurrent execution does not exist on most computers, when writing multithreaded programs, you should assume that it does. This is because you can't know the precise order in which separate threads will be executed,

or if they will execute in the same sequence twice. Thus, its best to program as if true concurrent execution is the case.

A.2 Multithreading Changes the Architecture of a Program

Multithreading changes the fundamental architecture of a program. Unlike a single-threaded program that executes in a strictly linear fashion, a multithreaded program executes portions of itself concurrently. Thus, all multithreaded programs include an element of parallelism. Consequently, a major issue in multithreaded programs is managing the interaction of the threads.

As explained earlier, all processes have at least one thread of execution, which is called the main thread. The main thread is created when your program begins. In a multithreaded program, the main thread creates one or more child threads. Thus, each multithreaded process starts with one thread of execution and then creates one or more additional threads. In a properly designed program, each thread represents a single logical unit of activity.

The principal advantage of multithreading is that it enables you to write very efficient programs because it lets you utilize the idle time that is present in most programs. Most I/O devices, whether they are network ports, disk drives, or the keyboard, are much slower than the CPU. Often, a program will spend a majority of its execution time waiting to send or receive data. With the careful use of multithreading, your program can execute another task during this idle time. For example, while one part of your program is sending a file over the Internet, another part can be reading keyboard input, and still another can be buffering the next block of data to send.

A.3 Thread Basics

- Thread operations include thread creation, termination, synchronization (joins,blocking), scheduling, data management and process interaction.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.
- All threads within a process share the same address space.

- Threads in the same process share:
    - Process instructions
    - Most data
    - open files (descriptors)
    - signals and signal handlers
    - current working directory
    - User and group id
- Each thread has a unique:
    - Thread ID
    - set of registers, stack pointer
    - stack for local variables, return addresses
    - signal mask
    - priority
    - Return value: errno
- pthread functions return "0" if OK.

A.4 Thread Creation and Termination

Example: pthread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

void main()
{
    pthread_t thread1, thread2;
    char *message1 = "Thread 1";
    char *message2 = "Thread 2";
    int  iret1, iret2;
```

```
    /* Create independent threads each of which will execute function */

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

    /* Wait till threads are complete before main continues. Unless we  */
    /* wait we run the risk of executing an exit which will terminate   */
    /* the process and all threads before the threads have completed.   */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}
```

Results:

```
        Thread 1
        Thread 2
        Thread 1 returns: 0
        Thread 2 returns: 0
```

Details:

- In this example the same function is used in each thread. The arguments are different. The functions need not be the same.
- Threads terminate by explicitly calling pthread_exit, by letting the function return, or by a call to the function exit which will terminate the process including any threads.
- Function call: pthread_create

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

Arguments:

- thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- attr - Set to NULL if default thread attributes are used. (else define members of the struct pthread_attr_t defined in bits/pthreadtypes.h) Attributes include:
  - detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
  - scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
  - scheduling parameter
  - inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)

        - scope (Kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
        - guard size
        - stack address (See unistd.h and bits/posix_opt.h _POSIX_THREAD_ATTR_STACKADDR)
        - stack size (default minimum PTHREAD_STACK_SIZE set in pthread.h),
    - void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.
    - *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

- Function call: pthread_exit

  void pthread_exit(void *retval);

  Arguments:
    - retval - Return value of thread.

      This routine kills the thread. The pthread_exit function never returns. If the thread is not detached, the thread id and return value may be examined from another thread by using pthread_join. Note: the return pointer *retval, must not be of local scope otherwise it would cease to exist once the thread terminates.

A.5 Thread Synchronization

The threads library provides three synchronization mechanisms:

- mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.
- joins - Make a thread wait till others are complete (terminated).
- condition variables - data type pthread_cond_t

Condition Variables

A condition variable is a variable of type pthread_cond_t and is used with the appropriate functions for waiting and later, process continuation. The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true. A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it resulting in a deadlock. The thread will be perpetually waiting for a signal that is never sent. Any mutex can be used, there is no explicit link between the mutex and the condition variable.

Functions used in conjunction with the condition variable:

- Creating/Destroying:
    - pthread_cond_init
    - pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
    - pthread_cond_destroy
- Waiting on condition:
    - pthread_cond_wait
    - pthread_cond_timedwait - place limit on how long it will block.
- Waking thread based on condition:
    - pthread_cond_signal
    - pthread_cond_broadcast - wake up all threads blocked by the specified condition variable.

Appendix B

SIMD Instructions: SSE, MMX, etc.

An increasing number of newer processors come with extensions to their instruction set commonly referred to as SIMD instructions. SIMD stands for Single Instruction Multiple Data meaning that a single instruction, such as "add", operates on a number of data items in parallel. A typical SIMD instruction for example will add 8 16 bit values in parallel. Obviously this can increase execution speed dramatically which is why these instruction set extensions were created.[21]

B.1 MMX Instruction Set

In 1997 Intel launched the Pentium Processor with MMX Technology. Subsequently the Pentium II, III and 4 were launched and other processors, such as the AMD Athlon and Duron, added this instruction set extension.

The MMX instruction set introduces 8 new 64 bit wide registers named mm0 .. mm7; however, these are mapped on the FPU registers, so FPU and MMX instructions can not be intermixed. The registers can hold the following data types:

Table B.1 MMX Registers

Data type	MMX-Register							
1x 64 bit (raw) integer	Bits 63 .. 0							
2x 32 bit integers	32 bit word 1				16 bit word 0			
4x 16 bit integers	16 bit word 3		16 bit word 2		16 bit word 1		16 bit word 0	
8x 8 bit integers	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0

Corresponding integer instructions were introduced:

Table B.2 MMX Instruction set

Instruction class	Instruction	Operation
Data movement	movq	64 bit move from/to mmx register
	movd	32 bit move from/to mmx register
Data format conversion / movement	packsswb/dw	Parallel pack signed words into bytes / dwords into words with signed saturation
	packuswb/dw	Parallel pack unsigned words into bytes / dwords into words with unsigned saturation
	punpckhbw/wd/dq	Parallel unpack high 32 bits: bytes to words / words to dwords / dwords to qwords
	punpcklbw/wd/dq	Parallel unpack low 32 bits: bytes to words / words to dwords / dwords to qwords
Arithmetical operations	paddb/w/d	Parallel add for bytes / words / dwords
	psubb/w/d	Parallel subtraction for bytes / words / dwords
	paddsb/w	Parallel saturated add for signed bytes / words
	paddusb/w	Parallel saturated add for unsigned bytes / words
	psubsb/w	Parallel saturated subtraction for signed bytes / words
	psubusb/w	Parallel saturated subtraction for unsigned bytes / words
	pmaddwd	Parallel multiply signed words and add the results

Table B.2 MMX Instruction set (con.)

Instruction class	Instruction	Operation
Arithmetical operations	pmullhw	Parallel multiply signed words and store high 16 bits of results
	pmulllw	Parallel multiply signed words and store low 16 bits of results
	pcmpeqb/w/d	Parallel compare signed bytes / words / dwords for equality
	pcmpgtb/w/d	Parallel compare signed bytes / words / dwords for "greater than"
Logical operations	pand	Parallel and operation on all 64 bits
	pandn	Parallel and not operation on all 64 bits
	por	Parallel or operation on all 64 bits
	pxor	Parallel xor operation on all 64 bits
Shift operations	psllw/d/q	Parallel shift logical left words / dwords / qwords
	psraw/d	Parallel shift right signed words / dwords
	psrlw/d/q	Parallel shift right unsigned words / dwords / qwords
Enable FPU instructions	emms	Resets the FPU register states to enable FPU instructions

All operations are either register, register or register, memory operations. No memory, register or memory, memory operations are available.

Here come some examples:

Table B.3 Examples of MMX Instruction set

Instruction	Operation
pxor mm0,mm0	Clear mm0
paddb mm0,mm1	Parallel add bytes from mm1 to mm0
psubusw mm7,[eax]	Parallel add unsigned words with saturation from [eax] to mm7
psslw mm5,3	Parallel shift left word in mm5 by 3 positions, filling in zeros

As can be seen from the above instruction list there are a number of limitations on this initial set of instructions:

- Not all data types are supported for all operations, i.e. unsigned multiplication, byte-wide shifts are missing etc.
- Various interesting operations are missing, i.e. min/max
- The registers are shared with the FPU registers, prohibiting any FPU operations in parallel
- No floating point operations are possible
- The registers are only 64 bits wide
- There are only 8 registers
- There are not prefetch or write-through instructions
- There are no real provisions for unaligned access
- Constants are not supported

The subsequent introduction of the SSE and SSE2 instruction set extensions remove many of these limitations.

B.2 SSE Instruction Set

The Pentium III processor, introduced in 1999, was the first processor with the SSE instruction set extension. However, AMD at that time already had added the similar 3DNow! instruction set extension to their AMD K6 CPU.

The SSE instruction set in fact consists of 3 distinct enhancements:

- Additional MMX instructions such as min/max
- Prefetch and write-through instructions for optimizing data movement from and to the L2/L3 caches and main memory
- New 128 bit XMM registers and corresponding 32 bit floating point (single precision) instructions

The SSE instruction set introduces 8 new 128 bit wide registers named xmm0 … xmm7. The registers can hold the following data types:

Table B.4 SSE Resgisters

Data type	XMM-Register			
1x 128 bit raw integer	Bits 127 ... 0			
4x 32 bit floating point values	32 bit floating point value 3	32 bit floating point value 2	32 bit floating point value 1	32 bit floating point value 0

The SSE instruction set includes the MMX instruction set; however, the following table lists only the additional SSE instructions:

Table B.5 SSE Instruction set

Instruction class	Instruction	Operation
Prefetch and write-through instructions	prefetchx	Prefetch cache line into L1/L2/L3 cache
	movntq	Store data directly into main memory, bypassing the caches (MMX only)
	movntps	Store data directly into main memory, bypassing the caches (XMM only)
	maskmovq	Store individual bytes directly into main memory, bypassing the caches (MMX only)

Table B.5 SSE Instruction set (con.)

Instruction class	Instruction	Operation
XMM data movement	movaps	128 bit move from/to xmm register, memory data must be aligned
	movups	128 bit move from/to xmm register, memory data may be unaligned
	movss	32 bit move from/to xmm register
	movlps	64 bit move from/to lower 64 bit of xmm register
	movhps	64 bit move from/to high 64 bit of xmm register
	movlhps	64 bit move from lower 64 bit of xmm register to high 64 bit of xmm register
	movhlps	64 bit move from higher 64 bit of xmm register to lower 64 bit of xmm register
	movmskps	Extract upper bits of all dwords and store in a general register
Data format conversion / movement	pextrw	Extract word into general register
	pinsrw	Insert word from general register
	pmovmskb	Extract upper bits of all bytes and store in a general register
	pshufw	Shuffle words (MMX only)
	pshufps	Shuffle dwords (XMM only)
	unpckhps	Parallel unpack and interleave high dwords (XMM only)
	unpcklps	Parallel unpack and interleave low dwords (XMM only)

Table B.5 SSE Instruction set (con.)

Instruction class	Instruction	Operation
Data format conversion / movement	cvtpi2ps	Parallel convert integer dwords to single precision floating point values (XMM only)
	cvtpi2ss	Convert integer dword to single precision floating point value (XMM only)
	cvtps2si	Parallel convert single precision floating point values to integer dwords (XMM only)
	cvtss2si	Convert single precision floating point value to integer dword (XMM only)
MMX arithmetical operations	pavgb/w	Parallel average for unsigned bytes / words
	pminub	Parallel minimum for unsigned bytes
	pmaxub	Parallel maximum for unsigned bytes
	pminsw	Parallel minimum for signed words
	pmaxsw	Parallel maximum for signed words
	pmulhuw	Parallel multiply unsigned words and store high 16 bits of results
	psadbw	Parallel absolute difference of unsigned bytes and sum up the results
XMM arithmetical operations	addps	Parallel add single precision floating point values
	addss	Add single precision floating point value in lower 32 bits
	subps	Parallel subtract single precision floating point values
	subss	Subtract single precision floating point value in lower 32 bits

Table B.5 SSE Instruction set (con.)

Instruction class	Instruction	Operation
XMM arithmetical operations	mulps	Parallel multiply single precision floating point values
	mulss	Multiply single precision floating point value in lower 32 bits
	divps	Parallel divide single precision floating point values
	divss	Divide single precision floating point value in lower 32 bits
	rcpps	Parallel approximate reciprocal single precision floating point values
	rcpss	Approximate reciprocal single precision floating point value in lower 32 bits
	sqrtps	Parallel square root single precision floating point values
	sqrtss	Square root single precision floating point value in lower 32 bits
	rsqrtps	Parallel approximate reciprocal square root single precision floating point values
	rsqrtss	Approximate reciprocal square root single precision floating point value in lower 32 bits
	minps	Parallel minimum of single precision floating point values
	minss	Minimum of precision floating point value in lower 32 bits
	maxps	Parallel maximum of single precision floating point values

Table B.5 SSE Instruction set (con.)

Instruction class	Instruction	Operation
XMM arithmetical operations	maxss	Maximum of precision floating point value in lower 32 bits
	cmpps	Parallel ompare single precision floating point values and return masks as result
	cmpss	Compare single precision floating point value in lower 32 bits and return mask as result
	comiss	Compare single precision floating point value in lower 32 bits and return result in flag register
	ucomiss	Compare unordered single precision floating point value in lower 32 bits and return result in flag register
XMM Logical operations	andps	Parallel and operation on all 128 bits
	andnps	Parallel and not operation on all 128 bits
	orps	Parallel or operation on all 128 bits
	xorps	Parallel xor operation on all 128 bits

Here come some examples:

Table B.6 Examples of SSE Instruction set

Instruction	Operation
prefetchnta [esi+64]	Prefetch data into L1 (or L2) cache from [esi+64]
pminub mm3,mm4	Parallel minimum of unsigned bytes of mm3 and mm4
mulps xmm0,xmm5	Parallel single precision floating point add xmm5 to xmm0
maxps xmm1,[edi]	Parallel single precision floating point maximum of xmm1 and data at [edi]

As can be seen from the above instruction list there are still a number of limitations on this set of instructions:

- Not all data types are supported for all operations, i.e. 32 bit integer multiplies, byte-wide shifts are missing etc.
- Only 32 bit floating point data types are available
- There are no integer instructions for the XMM registers
- There are only 8 registers
- There are no real provisions for unaligned access
- Constants are not supported

The subsequent introduction of the SSE2 instruction set extension removes some of these limitations.

B.3 SSE2 Instruction Set

The Pentium 4 processor introduced the SSE2 instruction set extensions in 2000.

Two important improvements were made:

- Support 64 bit (double precision) floating point data types
- Integer instruction for the XMM registers

Because the SSE2 instruction set introduces integer instructions to the XMM registers and double precision floating data types the XMM registers can now hold following data types:

Table B.7 SSE2 Registers

Data type	XMMx-Register															
1x 128 bit raw integer	Bits 127 .. 0															
2x 64 bit integer	64 bit word 1								64 bit word 0							
4x 32 bit integer	32 bit word 3				32 bit word 2				32 bit word 1				32 bit word 0			
8x 16 bit integer	16 bit word 7		16 bit word 6		16 bit word 5		16 bit word 4		16 bit word 3		16 bit word 2		16 bit word 1		16 bit word 0	
16x 8 bit integer	Byte 15	Byte 14	Byte 13	Byte 12	Byte 11	Byte 10	Byte 9	Byte 8	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
2x 64 bit floating point values	64 bit floating point value 1								64 bit floating point value 0							
4x 32 bit floating point values	32 bit floating point value 3				32 bit floating point value 2				32 bit floating point value 1				32 bit floating point value 0			

The SSE2 instruction set includes the SSE instruction set (the MMX and SSE integer instructions also apply to the XMM registers now and the single precision floating point instructions of the SSE instruction set also apply to the double precision data types); the following table lists only the additional SSE2 instructions:

Table B.8 SSE2 Instruction set

Instruction class	Instruction	Operation
Prefetch and write-through instructions	movntdq / movntpd	Store data directly into main memory, bypassing the caches (XMM only)
	movnti	Store data in general register directly into main memory, bypassing the caches
	maskmovdqu	Store individual bytes directly into main memory, bypassing the caches (XMM only)
XMM data movement	movapd / movdqa	128 bit move from/to xmm register, memory data must be aligned
	movupd / movdqu	128 bit move from/to xmm register, memory data may be unaligned
	movsd	64 bit move from/to xmm register
	movlpd	64 bit move from/to lower 64 bit of xmm register
	movhpd	64 bit move from/to high 64 bit of xmm register
	movlhpd	64 bit move from lower 64 bit of xmm register to high 64 bit of xmm register
	movhlpd	64 bit move from higher 64 bit of xmm register to lower 64 bit of xmm register
	movmskpd	Extract upper bits of all qwords and store in a general register
Data format conversion / movement	pmovmskpd	Extract upper bits of all qwords and store in a general register
	pshuflw	Shuffle words in lower 64 bits (XMM only)

Table B.8 SSE2 Instruction set (con.)

Instruction class	Instruction	Operation
Data format conversion / movement	pshufhw	Shuffle words in high 64 bits (XMM only)
	pshufpd	Shuffle dwords (XMM only)
	unpckhpd	Parallel unpack and interleave high qwords (XMM only)
	unpcklpd	Parallel unpack and interleave low qwords (XMM only)
	cvtpi2pd	Parallel convert integer dwords to double precision floating point values (XMM only)
	punpckhqdq	Parallel unpack high qwords
	punpcklqdq	Parallel unpack lower qwords
	cvtpi2sd	Convert integer dword to double precision floating point value (XMM only)
	cvtpd2si	Parallel convert double precision floating point values to integer qwords (XMM only)
	cvtsd2si	Convert double precision floating point value to integer qword (XMM only)
	cvtps2pd	Parallel convert single precision floating point values to double precision floating point values
	cvtpd2ps	Parallel convert double precision floating point values to single precision floating point values
	cvtss2sd	Convert single precision floating point value to double precision floating point value
	cvtsd2ss	Convert double precision floating point value to single precision floating point value

Table B.8 SSE2 Instruction set (con.)

Instruction class	Instruction	Operation
Data format conversion / movement	cvtps2dq	Parallel convert single precision floating point values to signed dword integers
	cvtdq2ps	Parallel convert signed dword integers to single precision floating point values
	movq2dq	Move MMX register value to XMM register
	movdq2q	Move lower 64 bit of XMM register to MMX register
Integer arithmetical operations	paddq	Parallel add for 64 bit integers
	psubq	Parallel subtraction for 64 bit integers
	pmuludq	Parallel multiply 32 bit unsigned integers and return 64 bit result
XMM arithmetical operations	addpd	Parallel add double precision floating point values
	addsd	Add double precision floating point value in lower 64 bits
	subpd	Parallel subtract double precision floating point values
	subsd	Subtract double precision floating point value in lower 64 bits
	mulpd	Parallel multiply double precision floating point values
	mulsd	Multiply double precision floating point value in lower 64 bits

Table B.8 SSE2 Instruction set (con.)

Instruction class	Instruction	Operation
XMM arithmetical operations	divpd	Parallel divide double precision floating point values
	divsd	Divide double precision floating point value in lower 64 bits
	rcppd	Parallel approximate reciprocal double precision floating point values
	rcpsd	Approximate reciprocal double precision floating point value in lower 64 bits
	sqrtpd	Parallel square root double precision floating point values
	sqrtsd	Square root double precision floating point value in lower 64 bits
	rsqrtpd	Parallel approximate reciprocal square root double precision floating point values
	rsqrtsd	Approximate reciprocal square root double precision floating point value in lower 64 bits
	minpd	Parallel minimum of double precision floating point values
	minsd	Minimum of precision floating point value in lower 64 bits
	maxpd	Parallel maximum of double precision floating point values
	maxsd	Maximum of precision floating point value in lower 64 bits

Table B.8 SSE2 Instruction set (con.)

Instruction class	Instruction	Operation
XMM arithmetical operations	cmpps	Compare double precision floating point values and return masks as result
	cmpsd	Compare double precision floating point value in lower 64 bits and return mask as result
	comisd	Compare double precision floating point value in lower 64 bits and return result in flag register
	ucomisd	Compare unordered double precision floating point value in lower 64 bits and return result in flag register
XMM logical operations	andpd	Parallel and operation on all 128 bits
	andnpd	Parallel and not operation on all 128 bits
	orpd	Parallel or operation on all 128 bits
	xorpd	Parallel xor operation on all 128 bits
XMM shift operations	pslldq	Shift all 128 bits left
	psrldq	Shift all 128 bits rights, filling in zeros

Here come some examples:

Table B.9 Examples of SSE2 Instruction set

Instruction	Operation
paddd xmm0,xmm6	Parallel add integer dwords from xmm6 to xmm0
pmaddwd xmm2,[edx]	Parallel multiply signed integer words in xmm2 and at [edi] and add results
sqrtpd xmm0,xmm0	Parallel double precision floating point square root of values in xmm0

Table B.9 Examples of SSE2 Instruction set (con.)

Instruction	Operation
addpd xmm2,[ebx+16]	Parallel add double precision floating point values from [ebx+16] to xmm2
psrldq xmm5,1	Shift right all 128 bits by one, filling in a zero

As can be seen from the above instruction list there are still a number of limitations on this set of instructions:

- Not all data types are supported for all operations, i.e. 32 bit saturated adds, byte-wide shifts are missing etc.
- There are only 8 registers
- There are no real provisions for unaligned access
- Constants are not supported

It is unclear whether future instruction set extensions will remove some of these limitations.

Nevertheless the SSE2 instruction set in effect is a parallel version of the typical integer and floating point instructions found in the basic instruction set of most processors, thereby enabling impressive speedups.

B.4 General Issues with SIMD Instructions

The basic characteristic of SIMD instructions is that they operate on n data items in parallel, n typically being 1, 2, 4, 8 or 16. There are mainly 5 problems with this:

- If the data layout does not match the SIMD requirements SIMD instructions can either not be used or data rearrangement code is necessary
- In case of unaligned data the performance will suffer dramatically; this problem leads to massively more complicated alignment code being necessary

- If the problem does not match the possible values of n, for example in case of a filter which would require n = 5, the solution is rather complicated
- (Parallel) Table lookups are not possible
- Compare operations are limited

Basically there are solutions to these problems, but they require some clever or additional code. Due to these problems even compilers which generate SIMD code will in many cases not do so or not be able to do so.

Appendix C Publications

[1] K. Chaichoompu, S. Kittitornkun, and S. Tongsima, “MT-ClustalW: Multithreading multiple sequence alignment”, Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006): Workshop on High Performance Computational Biology (HiCOMB06), 2006, Rhodes Island, Greece

[2] K. Chaichoompu and S. Kittitornkun, “Multithreaded ClustalW with Improved Optimization for Intel Multi-core Processor”, International Symposium on Communications and Information Technologies, 2006, Bangkok, Thailand

[3] K. Chaichoompu, S. Kittitornkun, and S. Tongsima, “Performance comparison of Multifactor Dimensionality Reduction: JVM vs. Native code”, International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC2006), 2006, Chiang Mai, Thailand

[4] K. Chaichoompu and S. Kittitornkun, “Optimized Multithreading Multiple Sequence Alignment for Intel Dual-core Processor”, 29th Electrical Engineering Conference (EECON #29), 2006, Pattaya, Thailand

[5] K. Chaichoompu and S. Kittitornkun, “On Experiments of MPEG-4 Video Uplink Transmission via GPRS”, 28th Electrical Engineering Conference (EECON #28), 2005, Phuket, Thailand

[6] K. Chaichoompu and S. Kittitornkun, “Protocologic Analyzer”, 27th Electrical Engineering Conference (EECON #27), 2004, Kon Kaen, Thailand

HiCOMB 2006
Fifth IEEE International Workshop on High Performance Computational Biology

Tuesday, April 25, 2006
Rodos Palace Resort Hotel & Convention Center, Rhodes Island, Greece

International Parallel & Distributed Processing Symposium

(Held in conjunction with the International Parallel and Distributed Processing Symposium)

MT-ClustalW: Multithreading Multiple Sequence Alignment

Kridsadakorn Chaichoompu[1], Surin Kittitornkun[1], and Sissades Tongsima[2]

[1]Dept.of Computer Engineering
Faculty of Engineering
King Mongkut's Institute of Technology
Ladkrabang, Bangkok 10520 ,Thailand
{s7060809,kksurin}@kmitl.ac.th

[2]National Center for Genetic Engineering
and Biotechnology
113 Paholyothin, Klong 1, Klong Luang
Pathumtani 12120, Thailand
sissades@biotec.or.th

Abstract

ClustalW is the most widely used tool for aligning multiple protein or nucleotide sequences. The alignment is achieved via three stages: pairwise alignment, guide tree generation and progressive alignment. This paper analyzes and enhances a multithreaded implementation of ClustalW called ClustalW-SMP for higher throughput. Our goal is to maximize the degree of parallelism on multithreading ClustalW called MultiThreading-ClustalW (MT-ClustalW). As a result, bioinformatics laboratories are able to use this MT-ClustalW with much less energy consumption on multicore and SMP (Symmetric MultiProcessor) machines than that of PC clusters. The experiment results show that the MT-ClustalW framework can achieve a considerable speedup over the sequential ClustalW and original multithreaded ClustalW-SMP implementations.

1 Introduction

Multiple sequence alignment of many nucleotides or amino acids is an important tool in bioinformatics. The multiple sequence alignment technique identifies diagnostic patterns or motif to characterize protein families. It can also detect or demonstrate homology between new sequences and existing families of sequences. Thus it helps predict the secondary and tertiary structures of the new sequences which is an essential prelude to molecular evolutionary analysis. Many multiple sequence alignment tools have been proposed to reduce the high computation time of fully performing alignment of all sequences. Implementations of various multiple sequence alignment heuristics include MSA [1], PRALINE [2], T-Coffee [3], DIALIGN P [4], MAFFT [5] and ClustalW [6]. ClustalW particularly is the most popular sequential program for multiple sequence alignment, and CLUSTALX [7] is a graphical interface version of ClustalW.

Caching and parallel methods are focused to improve the computation for the better throughput. The efficient execution of multiple sequence alignment method is concerned in the data server and the cluster of workstations about the effect of data caching. [8, 9] The parallel version of ClustalW is presented by SGI [10]. The SGI parallel version shows speedup of up to 10 folds when running ClustalW on 16 CPUs [11]. pCLUSTAL presents a parallel version of CLUSTALW. In contrast to the commercial SGI parallel Clustal version, which requires an expensive SGI multiprocessor system, pCLUSTAL can run on PC clusters as well [12]. ClustalW-MPI [13] is another interesting parallel ClustalW which uses a message-passing library called MPI and runs on distributed workstation clusters. Moreover, the parallel ClustalW with Dynamic Scheduling [14] proposes the algorithm that divides a progressive alignment into subtasks and schedules them dynamically. Besides the software approach, a new approach to MSA on reconfigurable hardware platforms to gain high performance at low cost. Fine-grained parallel processing elements (PEs) are designed for the computation of pairwise distances between protein sequences. [15]

This work present a parallel ClustalW algorithm using multithreading technique, called MT-ClustalW. Our achievement is considerable and done by dual-core processor which is much less expensive workstation than the high-end SGI multiprocessor.

This paper is organized as follows: Section 1 reviews the other parallel version of multiple sequence alignment tool, ClustalW. Section 2 describes the sequential ClustalW algorithm. Section 3 analyzes the SMP ver-

1-4244-0054-6/06/$20.00

sion of ClustalW. Section 4 presents the proposed parallel ClustalW, called MT-ClustalW. Section 5 demonstrates the experimental results. Finally, Section 6 draws the conclusion of this work.

2 Sequential ClustalW

ClustalW has become the most popular algorithm for multiple sequence alignment. This program implements a progressive method for multiple sequence alignment. As a progressive algorithm, ClustalW adds sequences one by one to the existing alignment to build a new alignment. The order of the sequences to be added to the new alignment is indicated by a precomputed phylogenetic tree, which is called a guide tree. The guide tree is constructed using the similarity of all possible pairs of sequences. The algorithm consists of 3 phases that are described below:

Stage 1 —Distance Matrix: All pairs of sequences are aligned separately in order to calculate a distance matrix based on the percentage of mismatches of each pair of sequences.

Stage 2 —Neighbor joining: The guide tree is calculated from the distance matrix using a neighbor joining algorithm [16]. The guide tree defines the order which the sequences are aligned in the next stage.

Stage 3 —Progressive alignment: The sequences are progressively aligned following the guide tree.

Now we discuss the complexity for all stages. Give N sequences and sequence length L, calculating the distance matrix in stage 1 takes $\mathcal{O}(N^2L^2)$ time. A neighbor joining algorithm is $\mathcal{O}(N^4)$ time for constructing the guide tree in stage 2. And for the last stage, progressive alignment is $\mathcal{O}(N^3+NL^2)$ time. The summary is shown in Table 1 [17].

3 Analysis : ClustalW-SMP

ClustalW-SMP (version 0.99–9) is the SMP version of ClustalW version 1.82, was created by O. Duzlevski [18]. We have analyzed ClustalW-SMP to find the bottleneck for improving the throughput and the speedup of the SMP version. Hence we used the Intel thread profiler tool [19] to analyze the SMP version. The profiler shows that the SMP version is not yet fully parallelized in Figure 1. This figure shows that only Distance matrix and Progressive alignment stages are forced into the threads, but not in Neighbor joining stage. Therefore, we modified the code of both the sequential ClustalW and the SMP versions and measure the execution time of three major stages. The two criteria to be analyzed are: first, the elapsed time of the three stages which we want to find the time ratio among the three stages and second, the number of threads which we want to find the relation between the thread number and the PC efficiency.

The protein sequence data sets from the National Center for Biotechnology Information (NCBI) are used as our test data whose sequence lengths and sequence numbers are shown in table 2. The profiles are done on 3GHz Intel Pentium IV processor with Hyper thread technology and 1 GB of memory. The elapsed times of ClustalW and ClustalW-SMP are compared which in this case ClustalW-SMP is faster. Although the executions are done in 2, 4 and 8 threads, the execution times of ClustalW-SMP still nearly, that shown in Table 3. Table 4 exhibits the elapsed times of each stage. In neighbor joining stage, the execution times are similar considering the same sequence numbers while the sequence lengths are fixed.

Figures 2 and 3 show the time ratios of the three stages of ClustalW-SMP with varying the number of sequences and the sequence lengths are fixed at 200 and 800 amino acids. The time ratios of neighbor joining stage with the fewer sequence numbers are more than the time ratios of neighbor joining stage with the more sequence numbers while the sequence lengths are fixed. That means if we can parallel the tasks in the neighbor joining stage, the execute time should be reduced more. This helps the alignment which needs to align the too many sequences to spend the faster time.

4 MT-ClustalW

We have modified the neighbor joining part of ClustalW-SMP. The implementation was done by using `pthread` library with `MUTEX` object as a synchronization object. Figure 4 shows the code for guide tree stage, the 4 nested loops has $\mathcal{O}(N^4)$ time complexity. We then break the 2 inner loops to be executed in parallel.

The flowchart of the thread synchronization is described in Figure 5. After the 2 inner loops were moved to be the parallel function, the parameters are set in the main thread. The main thread starts the parallel function by calling `signal(start)` and waits for the reply signal to execute next loop. The `thread_num` variable is the maximum number of thread that we can use. In parallel function, it executes parallel tasks after waiting for the start signal. Then the parallel

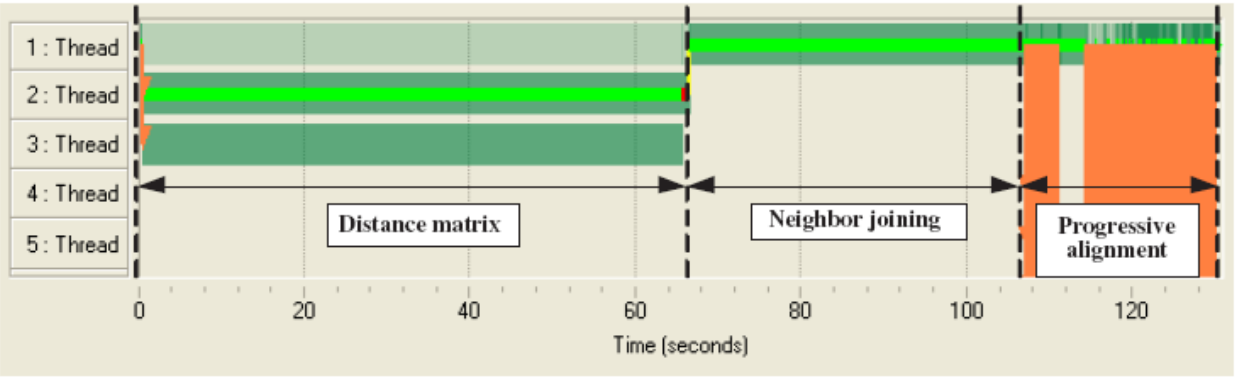

Figure 1. Intel Thread Profiler shows the time-line of the experiment with 400 protein sequences and about 200 amino acids in length .

Table 1. Complexity of the sequential ClustalW. The big-O asymptotic complexity of the elements of ClustalW as a function of L, the sequence length, and N, the number of sequences, retaining the highest-order terms in N with L fixed and vice versa.

Stage	Time complexity
Distance Matrix	$\mathcal{O}(N^2L^2)$
Neighbor joining	$\mathcal{O}(N^4)$
Progressive alignment	$\mathcal{O}(N^3 + NL^2)$
Total	$\mathcal{O}(N^4 + L^2)$

Table 2. Sequence length and Sequence number of the protein sequence as test data

Variables	Values
Average sequence lengths	200, 400, 600, 800
Sequence number	200, 400, 600, 800
Maximum thread number	2, 4, 8

Table 3. Elapsed times of ClustalW compare with ClustalW-SMP running in 2,4,8 threads

#seq-#len	ClustalW (*sec*)	ClustalW-SMP (*sec*)		
		2 Threads	4 Threads	8 Threads
200-200	26	22	22	22
200-800	494	420	414	415
400-200	130	118	116	116
400-800	1,932	1,667	1,665	1,676
600-200	496	469	462	462
600-800	4,544	3,943	3,939	3,959
800-200	1,395	1,347	1,341	1,341
800-800	8,539	7,500	7,480	7,520

Table 4. Elapsed times of ClustalW-SMP in each stage

# seq-# len	ClustalW-SMP (*sec*)		
	Distance Matrix	Neighbor Joining	Progressive Alignment
200-200	17	1	3
200-800	379	1	34
400-200	65	41	10
400-800	1,538	41	86
600-200	149	293	21
600-800	3,481	293	166
800-200	259	1,044	38
800-800	6,185	1,050	265

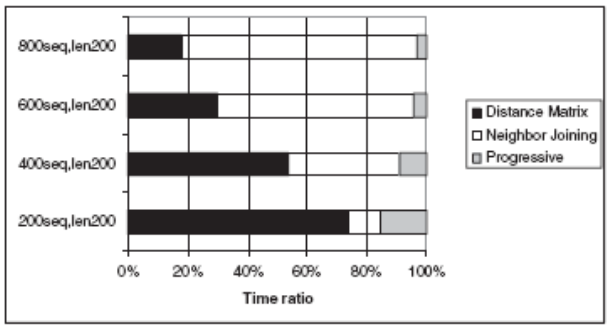

Figure 2. Time ratio of the three stages of ClustalW-SMP. The number of sequences are varied but the sequences lengths are fixed at about 200 amino acid

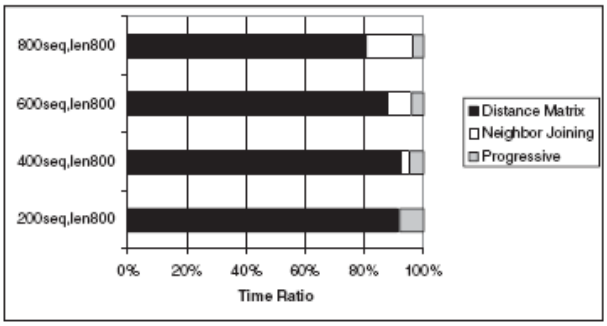

Figure 3. Time ratio of the three stages of ClustalW-SMP. The number of sequences are varied but the sequences lengths are fixed at about 800 amino acid

```
for(nc=1; nc<=(last_seq-first_seq+1-3); ++nc) {
        sumd = 0.0;
        for(j=2; j<=last_seq-first_seq+1; ++j)
        for(i=1; i<j; ++i) {
                tmat[j][i] = tmat[i][j];
                sumd = sumd + tmat[i][j];
                }
        tmin = 99999.0;
        for(jj=2; jj<=last_seq-first_seq+1; ++jj)
                if(tkill[jj] != 1)
                        for(ii=1; ii<jj; ++ii)
                                if(tkill[ii] != 1) {
                                        diq = djq = 0.0;
                                        for(i=1; i<=last_seq-first_seq+1; ++i) {
                                                diq = diq + tmat[i][ii];
                                                djq = djq + tmat[i][jj];
                                        }
                                        dij = tmat[ii][jj];
                                        d2r = diq + djq - (2.0*dij);
                                        dr  = sumd - dij -d2r;
                                        fnseqs2 = fnseqs - 2.0;
                                        total= d2r+ fnseqs2*dij +dr*2.0;
                                        total= total / (2.0*fnseqs2);
                                        if(total < tmin) {
                                                tmin = total;
                                                mini = ii;
                                                minj = jj;
                                        }
                                }
                        }
                        ...
```

THREAD BODY

Figure 4. The source code of the guide tree stage

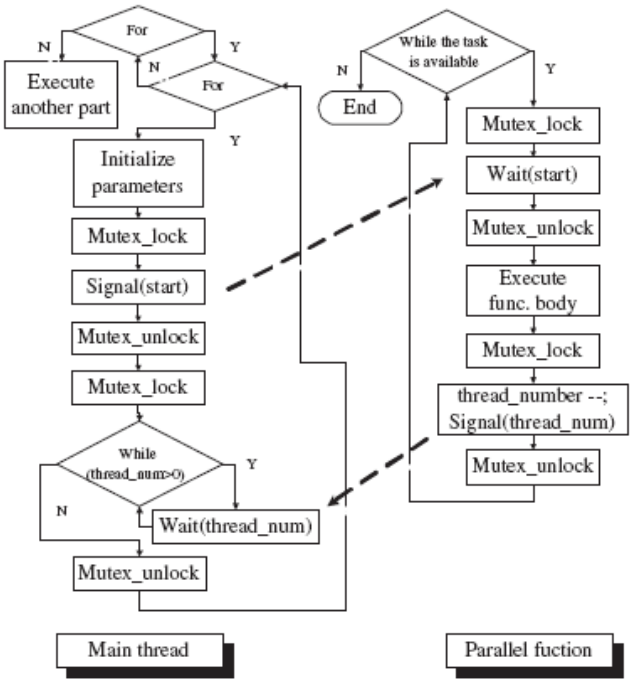

Figure 5. Flowchart of the thread synchronization

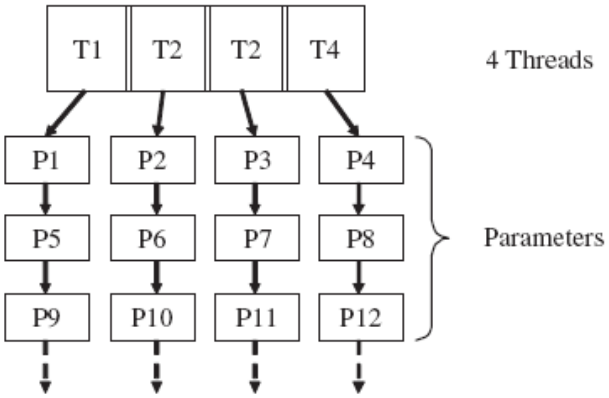

Figure 6. All loads are distributed to all thread functions equally as the parameters

function decreases the `thread_num` variable by one and send the `thread_num` signal to the main thread. After calling `signal(thread_num)`, the parallel function reaches the while condition and determine if there are more tasks to be executed. If so, the parallel function continues waiting for the start signal but if not, the function will cease. Back to the main thread, the main thread proceeds to the next loop after all parallel threads complete. When all loops are done, the main thread continues in the another part.

We reduce `pthread` creation overhead by creating the array of parallel functions. The set is equal to the maximum thread number that is set the user. All loads are distributed to all thread functions equally and taken by the parallel functions to execute the tasks. The logic diagram of the algorithm shows in Figure 6.

5 Experiment Results

We implement our algorithm in C programming language and utilize the `pthread` library. The experiment was conducted using a 2.8GHz Intel Pentium D (Dual-core) processor with 2 GB of memory. This computer runs MS Windows XP pro service pack 2 with no other applications installed. The elapsed times are measured as well as the wall clock time between the distance matrix stage and the progressive alignment stage. All measured times include the times that were taken to read the input data from a file and write the solution into a file. Furthermore, all measured times were measured when there is no one using the system except this experiment.

Our experiments measured the following: (1) The elapsed times as a function of the number of sequences in the guide tree stage. (2) Relative speedup (ratio of the elapsed time of ClustalW and the elapsed time of MT-ClustalW) as a function of the number of sequences.With the same test data set, Figure 7 shows the elapsed times for the ClustalW and MT-ClustalW results of the Neighbor joining stage as a function of number of sequences when running 8 threads. The MT-ClustalW successfully decreases the execution time. Figure 8 shows the speedup for the ClustalW-SMP and MT-ClustalW results of 8 threads as a function of number of sequences as compared to ClustalW. It can be observed that higher speedup can be achieved with the shorter sequence length to be aligned. When the sequence length is small, the speedup increase. When the sequence length is large, the speedup will decrease because the elapsed time of the guide tree stage depend on the number of sequences but the other stages depend on both the number of sequences and the sequence length. If the number of sequences is large and the sequence length is long, the ratio of the elapsed

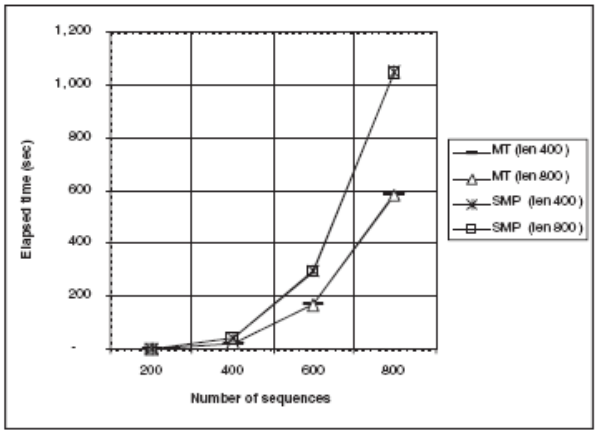

Figure 7. Elapsed times for the ClustalW and MT-ClustalW results of Neighbor joining stage as a function of number of sequences (8 threads)

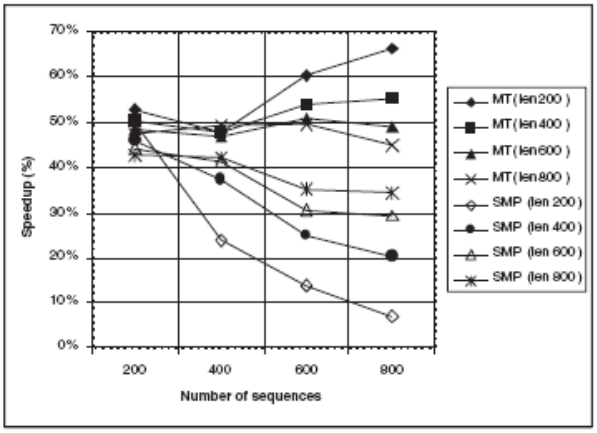

Figure 8. Speedup for the ClustalW-SMP and MT-ClustalW results of 8 threads as a function of number of sequences. Both speedup are compared with ClustalW

time of the guide tree stage and the elapsed time of the other stages is lower so the speedup decrease and vice versa.

Now, we will focus on the speedup of MT-ClustalW as a function of number of threads. In Figure 9, the speedup increases from 2 threads to 4 threads then decreases after 4 threads. Also in Figure 10, the speedup is almost the same as the speedup of Figure 9 except the 800 sequence data. The speedup seems to be lower then steady when the thread number is raised because the limitation of the machine resources.

The proposed method fully utilizes the multithreading feature in ClutalW and achieves the better speedup of ClustalW. MT-ClustalW is faster than ClustalW-SMP especially when the number of sequences is large. This will be compatible with the massive work for the alignment and gains more throughput.

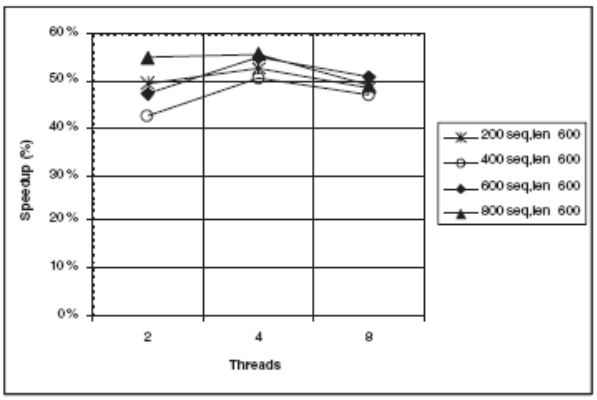

Figure 9. Speedup of the MT-ClustalW results as a function of number of threads that compared to ClustalW. The sequence lengths are fixed at 600 amino acids

6 Conclusion

In this paper, we presented a fully multithreading version of ClustalW called MT-ClustalW which significantly improves the performance of the traditional ClustalW. In the proposed MT-ClustalW, all stages: the distance matrix, the guide tree and the progressive alignment; are divided into a number of threads and executed on a Pentium-D machine. The proposed algo-

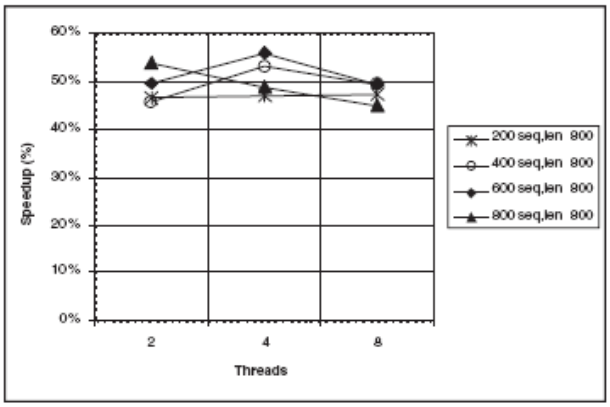

Figure 10. Speedups of the MT-ClustalW results as a function of number of threads that compared to ClustalW. The sequence lengths are fixed at 800 amino acids

rithm shows better speedups than the ClustalW-SMP and much better than that of sequential ClustalW.

References

[1] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu, "A tool for multiple sequence alignment," *Proc. Nail. Acad. Sci. USA*, vol. 86, pp. 4412–4415, 1989.

[2] V. A. Simossis and J. Heringa, "Praline: a multiple sequence alignment toolbox that integrates homology-extended and secondary structure information," *Nucleic Acids Research, 2005*, vol. 33, pp. 289–294, 2005.

[3] C. Notredame, D. G. Higgins, and J. Heringa, "T-coffee: A novel method for fast and accurate multiple sequence alignment," *IDEAL J. Mol. Biol. (2000)*, vol. 302, pp. 205–217, 2000.

[4] M. Schmollinger, K. Nieselt, M. Kaufmann, and B. Morgenstern, "Dialign p: Fast pair-wise and multiple sequence alignment using parallel processors," *BMC Bioinformatics 2004*, vol. 5, no. 128, 2004.

[5] K. Katoh, K. Misawa, K. Kuma, and T. Miyata, "Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform," *Nucleic Acids Research, 2002*, vol. 30, no. 14, pp. 3059–3066, 2002.

[6] J.D. Thompson, D.G. Higgins, and T.J. Gibson, "Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice," *Nucleic Acids Research 22*, vol. 22, no. 22, pp. 4673–4680, 1994.

[7] J. D. Thompson, T. J. Gibson, F. Plewniak, F. Jeanmougin, and D. G. Higgins, "The clustal_x windows interface: flexible strategies for multiple sequence alignment aided by quality analysis tools," *Nucleic Acids Research, 1997*, vol. 25, no. 24, pp. 48764882, 1997.

[8] U. Catalyurek, E. Stahlberg, R. Ferreira, and J. Saltz, "Improving performance of multiple sequence alignment analysis in multi-client environments," *Proceedings of the First IEEE Int. Workshop on High Performance Computational Biology (HiCOMB 2002, IPDPS 2002)*, 2002.

[9] U. Catalyurek, M. Gray, T. Kurc, J. Saltz, E. Stahlberg, and R. Ferreira, "A component-based implementation of multiple sequence alignment," *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC2003)*, pp. 122–126, 2003.

[10] http://www.sgi.com, "Silicon graphics, inc," .

[11] D. Mikhailov, Haruna C., and R. Gomperts, "Performance optimization of clustal w: Parallel clustal w, ht clustal, and multiclustal," *SGI ChemBio.*

[12] J. Cheetham, F. Dehne, S. Pitre, A. Rau-Chaplin, and P. J. Taillon, "Parallel clustal w for pc clusters," *Proceedings of International Conference on Computational Science and Its Applications (ICCSA)*, vol. 2668, pp. 300–309, 2003.

[13] K.B. Li, "Clustalw-mpi: Clustalw analysis using distributed and parallel computing," *Bioinformatics 19*, vol. 19, no. 12, pp. 1585–1586, 2003.

[14] J. Luo, I. Ahmad, M. Ahmed, and R. Paul, "Parallel multiple sequence alignment with dynamic scheduling," *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, 2005.

[15] D. Nathan, R. Clemens, T. Oliver, B. Schmidt, and D. Maskell, "Multiple sequence alignment

on an fpga," *Proceedings of the Forth IEEE Int. Workshop on High Performance Computational Biology (HiCOMB 2005, IPDPS 2005)*, 2005.

[16] N. Saitou and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees," *Mol Biol Evol*, vol. 4, no. 4, pp. 406–425, 1987.

[17] R.C. Edgar, "Muscle: a multiple sequence alignment method with reduced time and space complexity," *BMC Bioinformatics. 2004*, vol. 5, 2004.

[18] http://bioinfo.pbi.nrc.ca, "Clustalw_smp ver. 0.99-9," 2002.

[19] http://www.intel.com, "Intel thread profiler," .

IEEE

International Symposium on Communications and Information Technologies 2006 (ISCIT 2006)

October 18-20, 2006
Grand Mercure Fortune Hotel, Bangkok, Thailand

Office WEB site: www.telecom.kmitl.ac.th\iscit2006

ACCESS

Sponsored by:
NECTEC, Thailand
ECTI, Thailand
King Mongkut's Institute of Technology Ladkrabang, Thailand
IEEE Circuits and Systems Society
Technical Sponsored by: IEICE, Japan

IEEE Catalog Number: 06EX1447 ISBN: 0-7803-9741-X Library of Congress: 2006927941

Multithreaded ClustalW with Improved Optimization for Intel Multi-core Processor

Kridsadakorn Chaichoompu and Surin Kittitornkun
Dept.of Computer Engineering
Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Bangkok 10520 ,Thailand
E-mail: {s7060809,kksurin}@kmitl.ac.th

Abstract— **This paper presents the methodology that assists the compiler to optimize ClustalW; the most widely used tool for aligning multiple text-based protein or nucleotide sequences in Bioinformatics. Our goal is to minimize latency and maximize the throughput of execution on multithreading ClustalW called MT-ClustalW: our previous work. As a result, Optimized MT-ClustalW is able to fully utilize the machine resources and achieves higher throughput on multicore computers. The experiment results show that our methodology can assist the compiler to optimize the code better than only compiler-optimization and achieve over 2 times faster than the sequential ClustalW. Finally, we analyze the overall result with Amdahl's Law.**

I. Introduction

Multiple alignments of protein sequences are important in many applications, including phylogenetic tree estimation, structure prediction and critical residue identification. Many multiple sequence alignment tools such as MSA [1] and PRALINE [2], have been proposed to reduce the high computation time of fully performing alignment of all sequences. ClustalW particularly is the most popular sequential program for multiple sequence alignment, and CLUSTALX [3] is a graphical interface version of ClustalW.

Overview of parallel processing, a fast implementation of the Smith-Waterman sequence-alignment algorithm using Single-Instruction, Multiple-Data (SIMD) technology is presented. This implementation is based on the MultiMedia eXtensions (MMX) and Streaming SIMD Extensions (SSE) technology that is embedded in Intel's latest microprocessors. Six-fold speed-up relative to the fastest previously known Smith-Waterman implementation on the same hardware was achieved by an optimized 8-way parallel processing approach. [4] More about Smith-Waterman algorithm, a preliminary implementation of Smith-Waterman algorithm using a new chip multiprocessor architecture with multiple Digital Signal Processors (DSP) on a single chip leading to high performance at low cost. [5] Several methods of computing sequence alignments with limited memory per processing element on SIMD processing elements were presented in [6]. And this paper [7] improves alignment times by either reducing the alignment sensitivity or by developing specialized hardware. The solution comes in the form of parallel processing hardware such as Paracels GeneMatcher and Compugens Bioccelerator.

The parallel version of ClustalW is presented by SGI [8]. The SGI parallel version shows speedup of up to 10 folds when running ClustalW on 16 CPUs [9]. pCLUSTAL presents a parallel version of CLUSTALW. In contrast to the commercial SGI parallel Clustal version, which requires an expensive SGI multiprocessor system, pCLUSTAL can run on PC clusters as well [10]. ClustalW-MPI [11] is another interesting parallel ClustalW which uses a message-passing library called MPI and runs on distributed workstation clusters. Moreover, the parallel ClustalW with Dynamic Scheduling [12] proposes the algorithm that divides a progressive alignment into subtasks and schedules them dynamically. Besides the software approach, a new approach to MSA on reconfigurable hardware platforms to gain high performance at low cost. Fine-grained parallel processing elements (PEs) are designed for the computation of pairwise distances between protein sequences. [13] Caching and parallel methods are focused to improve the computation for the better throughput. The efficient execution of multiple sequence alignment method is concerned in the data server and the cluster of workstations about the effect of data caching. [14], [15]

This work presents the methods that assist the compiler to optimize the source code of ClustalW and MT-ClustalW and gain higher throughput. Our achievement is considerable and done by dual-core processor which is much less expensive workstation than the high-end multiprocessor.

This paper is organized as follows: Section I reviews the other version of multiple sequence alignment tool, ClustalW. Section II describes ClustalW and MT-ClustalW. Section II-A describes the sequential ClustalW algorithm. Section II-B describes the previous work, MT-ClustalW, which is the fully parallel version of ClustalW. Section III presents our methods for assisting the compiler in optimization. Section IV demonstrates the experimental results. Section V discusses about overall results. Finally, Section VI draws the conclusion of this work.

II. ClustalW and Multithreading

A. Sequential ClustalW

ClustalW has become the most popular algorithm for multiple sequence alignment. This program implements a progressive method for multiple sequence alignment. As a progressive

algorithm, ClustalW adds sequences one by one to the existing alignment to build a new alignment. The order of the sequences to be added to the new alignment is indicated by a pre-computed phylogenetic tree, which is called a guide tree. The guide tree is constructed using the similarity of all possible pairs of sequences. The algorithm consists of 3 phases that are described below:

Stage 1 — Distance Matrix: All pairs of sequences are aligned separately in order to calculate a distance matrix based on the percentage of mismatches of each pair of sequences.

Stage 2 — Neighbor joining: The guide tree is calculated from the distance matrix using a neighbor joining algorithm [16]. The guide tree defines the order which the sequences are aligned in the next stage.

Stage 3 — Progressive alignment: The sequences are progressively aligned following the guide tree.

B. MT-ClustalW

A fully multithreading version of ClustalW called MT-ClustalW [17] which significantly improves the performance of the traditional ClustalW. MT-ClustalW was modified from ClustalW-SMP and was done by using pthread library with MUTEX object as a synchronization object. In the proposed MT-ClustalW, all stages: the distance matrix, the guide tree and the progressive alignment; are divided into a number of threads and executed on a Pentium-D machine. So that bioinformatics laboratories are able to use this MT-ClustalW with much less energy consumption on multicore and SMP (Symmetric MultiProcessor) machines than that of PC clusters.

III. Improvement of MT-ClustalW (Enhanced version)

We used the Intel C++ Compiler for Windows [18] to optimize the MT-ClustalW by enabling the /QxP option which optimizes code for Intel Core Duo processors and Intel Core Solo processors, Intel Pentium 4 processors with Streaming SIMD Extensions 3, and compatible Intel processors with Streaming SIMD Extensions 3 (SSE3). With the /QxP option, the compiler automatically traces the code and optimizes the loops. As we have done, only the simple loops were optimized. So we had to profile and change some codes to assist the compiler for optimizing the loops.

We used the Intel VTune Performance Analyzer to profile ClustalW in debug mode and look for the hotspots within the functions. After profiling, we focus on the top-usage functions to search for the hotspots and modify the codes with our guide methodologies.

Here are the methodologies that we used and applied to the codes for assisting the compiler to optimize the code better than original ClustalW 1.8. The methodologies are described as follows: Loop reversal, Loop fission, Type Casting, and Reduce procedure calls. The profiling result and the applied methodologies were shown in Table I

A. Loop reversal

That is to run a loop backward. Reversal of `for` loops is always legal, since the execution is not defined in terms of the order of the index set. Thus, loop reversal is legal only when the loop carries no dependence relations. Here the example code, this loop is not vectorized by the compiler.

```
for (i=se2;i>0;i--)
{
    HH[i] = -1;
    DD[i] = -1;
}
```

After we reverse the loop, it can be vectorized as follows.

```
for (i=1;i<=se2;i++)
{
    HH[i] = -1;
    DD[i] = -1;
}
```

B. Loop fission

A single loop can be broken into two or more smaller loops. Loop fission can break up the block of conditionally executed statements. To apply loop fission, a temporary array must be used to hold the result of the variable which is not array. The temporary array is used to guard the execution of the statements after fission. The example code is shown as follow:

```
for (j=0;j<=N;j++)
{
    hh = HH[j] + RR[j];
    if (hh>=midh)
       if (HH[j]!=DD[j]&&RR[j]==SS[j])
       {
          midh=hh;
          midj=j;
       }
}
```

After applying loop fission, the code contains 2 loops. The first loop can be vectorized, but the second loop can not. However, we can make the compiler vectorize the second loop by using a temporary array to hold the result of the conditional test.

```
for (j=0;j<=N;j++) {
    temp[j] = HH[j] + RR[j];
}
for (j=0;j<=N;j++)
{
    if (temp[j]>=midh)
       if (HH[j]!=DD[j]&&RR[j]==SS[j])
       {
          midh=temp[j];
          midj=j;
       }
}
```

C. Type Casting

Converting an expression of a given type into another type. Type casting can be used to promote the variable from `integer` to `float`, this can help the compiler to vectorize the loop using the Intel MMX technology. The example is shown as follow:

```
for (ix=0;ix<=max_aa;ix++)
{
    score+=(profile1[n][ix]
           *profile2[m][ix]);
}
```

After applying this method, the loop is vectorized. However, don't be forget to convert the result of the statement to the original type.

```
for (ix=0;ix<=max_aa;ix++)
{
    score+=(int)((float)profile1[n][ix]
         * (float)profile2[m][ix]);
}
```

D. Procedure call reduction

Calling the procedure in the loop, the processor takes much overhead. The code can reduce the procedure calls by using `Macro`. The example code can be rewrite as `Macro` and is shown as follow:

```
static sint calc_score(sint iat,sint jat,
            sint v1,sint v2)
{
   sint ipos,jpos,ret;
   ipos = v1 + iat;
   jpos = v2 + jat;
   ret=matrix[(int)seq_array[seq1][ipos]]
       [(int)seq_array[seq2][jpos]];
   return(ret);
}
```

After rewriting as `Macro`, the execution time greatly reduces because there isn't need to process the procedure call during the nested loops and the recursions. The `Macro` code is like this:

```
#define calc_score(iat,jat,v1,v2)
   matrix[(int)seq_array[seq1][v1+iat]]
   [(int)seq_array[seq2][v2 + jat]]
```

IV. Experiment Results

We implement our method using Intel C++ Compiler 9.0. The experiment was conducted using a 2.8GHz Intel 820 Pentium D (Dual-core) processor with 2 GB of memory. This computer runs MS Windows XP pro service pack 2 with no other applications installed. We compared the results of MT-ClustalW with ClustalW and got the same results. The elapsed times are measured as well as the wall clock time between the distance matrix stage and the progressive alignment stage. All measured times exclude the times that were taken to read the input data from a file and write the solution into a file. Furthermore, all measured times were measured when there is no one using the system except this experiment.

Our experiments measured as follows: (1) The elapsed times as a function of the number of sequences in every stages. (2) The relative speedup (ratio of the elapsed times of both ClustalW and MT-ClustalW and the elapsed times of our assist version of both ClustalW and MT-ClustalW) as a function of the number of sequences. We have done the whole experiment for 3 times with the test data set as shows in Table II. We ran the MT-ClustalW with 4 threads as refer from our previous work [17]. With the different running mode, Table III shows the elapsed times of ClustalW and MT-ClustalW in each stage and the overall speedup. The elapsed times mainly reduce in the distance matrix stage for both ClustalW and MT-ClustalW, but little reduce in the neighbor Joining stage and the progressive Alignment stage. And the results are quite similar though the test data has the different size. With the same experiment, the speedups increase from Running mode II to Running mode VI, in order. All speedups are compared with the sequential ClustalW or Running mode I.

With the same test data set, Fig. 1 shows the speedups of the optimized versions of ClustalW as a function of number of sequences. All speedups are compared with original ClustalW. It can be observed that ClustalW which is optimized with our assist gains significantly higher speedup than only compiler-optimization. Also when the sequence length changes from 800 to 1000 amino acids, ClustalW which is optimized with our assist achieves higher speedup. However, the speedups of ClustalW of both versions decrease when the number of sequences is large. As same as ClustalW result, Fig. 2 shows the speedups of the optimized versions of MT-ClustalW, but MT-ClustalW which is optimized with our assist gains much higher speedup when the sequence length is fixed at 1000 amino acids. More about the results of MT-ClustalW, see the previous research in [17].

The proposed method fully utilizes the optimization feature in ClutalW and achieves the better speedup of ClustalW. MT-ClustalW is faster than ClustalW especially when the number of sequences is large. This will be compatible with the massive work for the alignment and gains more throughput.

V. Overall Results and Discussion

This section, we describe the result with Amdahl's Law. In essence, Amdahl's Law states that the overall speedup of a computer system depends on both the speedup in a particular component and how much that component is used by the system. In symbols:

$$S = \frac{1}{(1-f) + \frac{f}{k}} \tag{1}$$

where S is the overall system speedup, f is the fraction of work performed by the improved part, and k is the speedup of a new component.

TABLE I

PROFILING RESULT AND OPTIMIZATION METHODOLOGY OF CLUSTALW

Function	Clockticks (%)	Optimization methodology*
diff	33.36	A, B
prfscore	15.93	C
forward_pass	14.91	-
calc_score	12.93	D
reverse_pass	11.45	A
pdiff	5.85	-

*Note: A is Loop reversal, B is Loop fission, C is Type Casting, and D is Procedure call reduction

TABLE II

PARAMETERS OF THE EXPERIMENT AND THE TEST DATA SET

Parameter	Value
Number of experiments	3 times
Number of threads	4 threads
Sequence lengths	200, 400, 600, 800
Number of sequences	800, 1000

TABLE III

ELAPSED TIMES IN EACH STAGE AND OVERALL SPEEDUP OF CLUSTALW AND MT-CLUSTALW

Running mode*	Elapsed times (*ms*)			Overall speedup
	Distance Matrix	Neighbor Joining	Progressive Alignment	
Test data #1 - 200 sequences, 800 amino acids				
I	448,922	1,485	30,078	-
II	386,375	1,297	30,468	1.15
III	361,297	1,297	29,188	1.23
IV	264,172	1,234	19,859	1.68
V	220,312	1,063	19,781	1.99
VI	205,297	1,078	18,922	2.13
Test data #2 - 800 sequences, 1000 amino acids				
I	11,918,672	932,718	333,110	-
II	10,387,046	881,125	338,016	1.14
III	9,656,750	880,969	327,985	1.21
IV	7,009,875	511,047	252,984	1.70
V	5,900,891	473,359	253,188	1.98
VI	5,472,407	474,109	244,672	2.12

*Note: Running mode defines as follows: (I) ClustalW without optimization (II) ClustalW with optimization (III) ClustalW with optimization and our assist (IV) MT-ClustalW without optimization (V) MT-ClustalW with optimization (VI) MT-ClustalW with optimization and our assist

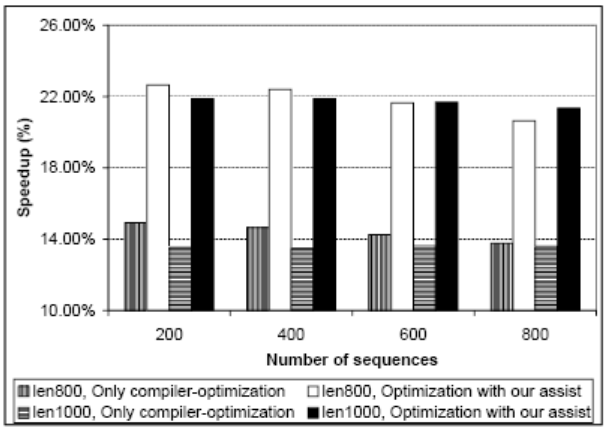

Fig. 1. Speedup of the optimized versions of ClustalW as a function of number of sequences. The sequence lengths are fixed at 800 and 1000 amino acids. All speedups are compared with the original sequential ClustalW

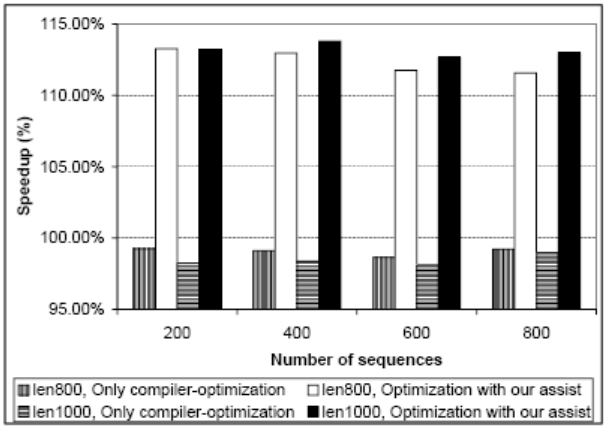

Fig. 2. Speedup of the optimized versions of MT-ClustalW as a function of number of sequences. The sequence lengths are fixed at 800 and 1000 amino acids. All speedups are compared with the original sequential ClustalW

For the vectorized optimization, we can approximate that k is 4 because the processor can execute 4 data for each instruction. As referred to (1), we can calculate f as:

$$f_{opt} = -\frac{4}{3}[\frac{1-S}{S}] \tag{2}$$

where f_{opt} is the improved fraction of the optimized version of ClustalW. For the experiment of ClustalW with fully optimization testing with 800 sequences (length is 1000 amino acids), the total speedup is 1.21 times as in table III, therefore we can find the improved fraction of ClustalW with optimization as in (2), we have:

$$f_{opt} = -\frac{4}{3}[\frac{1-1.21}{1.21}] = 0.23$$

So we can conclude that the fraction of the improved part is 23% for this experiment.

And for the fully multithread version of ClustalW running on the dual-core processors, we can approximate that k is 2. As referred to (1), we can calculate f as:

$$f_{mt} = -2[\frac{1-S}{S}] \tag{3}$$

where f_{mt} is the improved fraction of the multithreading ClustalW. With the same test data, we can find the improved fraction of the multithreading ClustalW as in (3), so we have:

$$f_{mt} = -2[\frac{1-1.70}{1.70}] = 0.82$$

We can conclude that the fraction of the improved part is 82% for this experiment.

Finally, the fully multithread version of ClustalW with optimization should gain as follow:

$$Speedup_{total} = Speedup_{opt} \times Speedup_{mt} \tag{4}$$

where $Speedup_{total}$ is the overall system speedup, $Speedup_{opt}$ is the speedup of ClustalW with optimization, and $Speedup_{mt}$ is the speedup of the multithreading ClustalW. As the example experiment, we can calculate the total speedup as in (4), so we get:

$$Speedup_{total} = 1.21 \times 1.70 = 2.06$$

As referred to the experiment result, the completed version of ClustalW achieve 2.13 times faster that relates to the above discussion.

VI. Conclusion

In this paper, we presented an optimized version of MT-ClustalW which significantly improves the performance of the traditional ClustalW. The optimization is done with our proposed methods that are able to assist the compiler in optimizing. The optimized MT-ClustalW is done for a Pentium-D machine by using the Intel C++ compiler. The proposed methods achieve better speedups than the original MT-ClustalW and much better than that of sequential ClustalW.

References

[1] D. J. Lipman, S. F. Altschul, and J. D. Kececioglu, "A tool for multiple sequence alignment," *Proc. Nail. Acad. Sci. USA*, vol. 86, pp. 4412–4415, 1989.

[2] V. A. Simossis and J. Heringa, "Praline: a multiple sequence alignment toolbox that integrates homology-extended and secondary structure information," *Nucleic Acids Research, 2005*, vol. 33, pp. 289–294, 2005.

[3] J. D. Thompson, T. J. Gibson, F. Plewniak, F. Jeanmougin, and D. G. Higgins, "The clustal_x windows interface: flexible strategies for multiple sequence alignment aided by quality analysis tools," *Nucleic Acids Research, 1997*, vol. 25, no. 24, pp. 48764882, 1997.

[4] T. Rognes and E. Seeberg, "Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics 2000*, vol. 16, no. 8, pp. 699–706, 2000.

[5] X. Meng and V. Chaudhary, "Bio-sequence analysis with cradles 3soc software scalable system on chip," *ACM Symposium on Applied Computing*, 2004.

[6] J.A. Grice, R. Hughey, and D. Speck, "Parallel sequence alignment in limited space," *Proceedings of International Conference Intelligent Systems for Molecular*, pp. 145–153, 1995.

[7] F. Lau, "An integrated approach to fast, sensitive, and cost-effective smith-waterman multiple sequence alignment," *Bioinformatics Module*, 2000.

[8] http://www.sgi.com, "Silicon graphics, inc," .

[9] D. Mikhailov, Haruna C., and R. Gomperts, "Performance optimization of clustal w: Parallel clustal w, ht clustal, and multiclustal," *SGI ChemBio*.

[10] J. Cheetham, F. Dehne, S. Pitre, A. Rau-Chaplin, and P. J. Taillon, "Parallel clustal w for pc clusters," *Proceedings of International Conference on Computational Science and Its Applications (ICCSA)*, vol. 2668, pp. 300–309, 2003.

[11] K.B. Li, "Clustalw-mpi: Clustalw analysis using distributed and parallel computing," *Bioinformatics 19*, vol. 19, no. 12, pp. 1585–1586, 2003.

[12] J. Luo, I. Ahmad, M. Ahmed, and R. Paul, "Parallel multiple sequence alignment with dynamic scheduling," *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, 2005.

[13] D. Nathan, R. Clemens, T. Oliver, B. Schmidt, and D. Maskell, "Multiple sequence alignment on an fpga," *Proceedings of the Forth IEEE Int. Workshop on High Performance Computational Biology (HiCOMB 2005, IPDPS 2005)*, 2005.

[14] U. Catalyurek, E. Stahlberg, R. Ferreira, and J. Saltz, "Improving performance of multiple sequence alignment analysis in multi-client environments," *Proceedings of the First IEEE Int. Workshop on High Performance Computational Biology (HiCOMB 2002, IPDPS 2002)*, 2002.

[15] U. Catalyurek, M. Gray, T. Kurc, J. Saltz, E. Stahlberg, and R. Ferreira, "A component-based implementation of multiple sequence alignment," *Proceedings of the 2003 ACM Symposium on Applied Computing (SAC2003)*, pp. 122–126, 2003.

[16] N. Saitou and M. Nei, "The neighbor-joining method: a new method for reconstructing phylogenetic trees," *Mol Biol Evol*, vol. 4, no. 4, pp. 406–425, 1987.

[17] K. Chaichoompu, S. Kittitornkun, and S. Tongsima, "Mt-clustalw: Multithreading multiple sequence alignment," *Proceedings of the IEEE IPDPS 2006 (HiCOMB06)*, 2006.

[18] http://www.intel.com, "Intel c++ compiler for windows," .

MENU
Home
Call for Paper
Committee
IACs
ICCs
TPCs
Keynote Speaker
Schedule
Proceedings Vol. I
Proceedings Vol. II
Proceedings Vol. III
Travel Infomation
Sightseeing
Contact Us

Bio-Computing

A Reconfigurable CMOS Power Amplifier by using Switched Network for ZigBee/Bluetooth Applications
Seok-Oh Yun and Hyung-Joun Yoo (Information and Communications Univ.)

Performance comparison of Multifactor Dimensionality Reduction: JVM vs. Native code
Kridsadakorn Chaichoompu, Surin Kittitornkun (King Mongkut's Inst. of Tech. Ladkrabang) and Sissades Tongsima (National Center for Genetic Engineering and Biotechnology)

Aligning Multiple Protein Sequences by An Evolutionary Tree-Base Method
Farhana Naznin, Morikazu Nakamura, Chihiro Miyazato, Takeo Okazaki, Yumiko Nakajima (Univ. of the Ryukyus)

A Method for Outlier Detection Based on Area Descent
Huynh Trung Hieu and Yonggwan Won (Chonnam National Univ.)

An Improvement of Linear Regression with Elimination of Outliers Detected by Area-Descent
Huynh Trung Hieu and Yonggwan Won (Chonnam National Univ.)

<<< Back >>>

PERFORMANCE COMPARISON OF MULTIFACTOR DIMENSIONALITY REDUCTION: JVM VS. NATIVE CODE

Kridsadakorn Chaichoompu[1], *Surin Kittitornkun*[1], *and Sissades Tongsima*[2]

[1]Dept.of Computer Engineering
Faculty of Engineering
King Mongkut's Institute of Technology
Ladkrabang, Bangkok 10520 ,Thailand
{s7060809,kksurin}@kmitl.ac.th

[2]National Center for Genetic Engineering
and Biotechnology
113 Paholyothin, Klong 1, Klong Luang
Pathumtani 12120, Thailand
sissades@biotec.or.th

ABSTRACT

Multifactor Dimensionality Reduction (MDR) was originally developed to detect gene-gene interactions in the domain of human genetics. MDR algorithm collapses high-dimensional genetic data into a single dimension thus permitting interactions to be detected in relatively small sample sizes. To deal with the large volume of data, we demonstrate the experiment of MDR software which was written in Java programming language and compare with native method. We focus on execution time, memory usage and CPU usage and describe the results.

1. INTRODUCTION

Executing the Java bytecode on any operating system (OS) platform requires a Java Virtual Machine (JVM) to be installed on the machine. There must be sufficient system resources (memory and disk space) to support the JVM. So that, compiling the Java code into native executable file can run at much improved speed and require significantly less memory and disk space. One Java native compiler, GCJ is a portable, optimizing, ahead-of-time compiler for the Java programming language. GCJ is part of the GNU Compiler Collection (GCC) [1] and it is free software. More commercial Java native compiler, Excelsior JET is a toolkit and complete runtime environment for acceleration, protection, and deployment of Java applications. Excelsior JET demonstrated better performance and lower resources demand in the test against two popular JVMs. [2] Moreover about native method, there is a optimizing translator from Java bytecode to native machine code. The major technical issues involve in stack to register mapping, run-time memory structure mapping, and exception handlers. Encouraging initial results based on our x86 port are presented. [3]

More experiment focusing on Java and the native libraries, the Java versions of parallel benchmarks from the ParkBench suite is run to evaluate the performance of Java code which accesses native libraries. [4] And there is the suggestion of architecture that uses Java native interface to improve the performance of Java domain transformation applications. [5]

This work presents a performance of MDR [6] software using JVM tuning technique and compare with native method. Our experiment is done by dual-core processor which will be commonly use in the nearly future.

This paper is organized as follows: Section 1 reviews the other Java applications relating native method. Section 2 describes the multifactor dimensionality reduction software, MDR. Section 3 describes the Java native compilation and tool. Section 4 demonstrates the experimental results. Finally, Section 5 draws the conclusion of this work.

2. MULTIFACTOR DIMENSIONALITY REDUCTION SOFTWARE : MDR

Multifactor dimensionality reduction (MDR) algorithm is a data mining approach for detecting and characterizing combinations of attributes or independent variables that interact to influence a dependent or class variable. MDR was designed specifically to identify interactions among discrete variables that influence a binary outcome and is considered a nonparametric alternative to traditional statistical methods such as logistic regression.

The basis of the MDR method is a constructive induction algorithm that converts two or more variables or attributes to a single attribute. This process of constructing a new attribute changes the representation space of the data. The end goal is to create or discover

ISBN 974-94418-9-3 ©2006 ECTI

The 21st International Technical Conference on Circuits/Systems, Computers and Communications

a representation that facilitates the detection of nonlinear or nonadditive interactions among the attributes such that prediction of the class variable is improved over that of the original representation of the data.

3. JAVA NATIVE COMPILATION

Compiling Java source code into Java bytecode, you must execute the results on any hardware/OS platform that has installed a JVM. Not only a JVM must be available for any platform on which you want to run your Java applications, but there must be significant system resources to support the JVM. As a result, many developers continue to rely on more targeted languages such as C/C++ programming language.

Compiling source code in C/C++ language is similar. Once the code is written, you run it through a compiler and linker targeted to a specific hardware/OS platform. The resulting application will be executable only on the targeted platform, but will not require that a JVM be installed.

Attempting to bridge the best of the above solutions, developers write applications in the Java programming language and compile them into native executable file. Once the Java code is written, it can be run through a Java compiler to produce Java bytecode, which is compiled into native code, or it can be run directly into a Java native compiler.

The advantage of this approach is that the executable file can be executed on the targeted platform without the JVM. This is intended to result in Java applications that execute at much improved speed and require significantly less memory and disk space.

Here is the tool that we used in our experiment, Excelsior JET [7]. It is a toolkit and complete runtime environment for optimizing, deploying and running applications written in the Java programming language. The Excelsior JET enables you to convert your application's classes and Java archive files (jar) into optimized Intel x86 code on the developer's system. As a result, you get high performance native executables for our system.

4. EXPERIMENT RESULTS

We compile the original MDR software, the source files in Java programming language, into native executable file using Excelsior JET version 4.1 evaluation. The experiment was conducted using a 2.8GHz Intel Pentium D (Dual-core) processor with 2 GB of memory. This computer runs MS Windows XP pro service pack 2 with no other applications installed. The execution times are measured as well as the wall clock time until finishing. All measured times include the times that were taken to read the input data from a file and write the solution to a screen. Furthermore, all measured times were measured when there is no one using the system except this experiment.

For the fair comparison, we set the parameter for both native and Java modes as table 1. As for the Java mode, all parameters are set at run-time execution for tuning a JVM. However, we set all parameters in Excelsior Jet before compiling into the native mode.

Our experiments measured the following: (1) The average execution times of MDR software both native and Java modes. (2) The percentage of processor time as function of time. (3) The page faults/*sec* as function of time. (4) The virtual memory usage as function of time. With the same test data set, we did the experiment for 5 times and the result of the average execution times are shown in table 2. The result shows that the Java mode achieves the faster execution time. Hence we used the performance tool (MS Windows XP Administrative tool) to profile both native and Java modes for more details. Figure 1 shows the percentage of processor time as function of elapsed time. The number of sequences are fixed at 20 sequences and the sequences lengths are varied. The running order of MDR modes and the sequences lengths are shown in table 3. Each running order is separated by 30 *sec* delay command. Both native and Java modes utilize the processor at about 50%, but the native mode is little higher. Same as figure 1, but we increase the number of sequences to be 30 sequences and the results are shown in figure 2.

With the memory profiling, figure 3 shows the virtual memory usage as function of elapsed time and the number of sequences are fixed at 20 sequences. Although the native modes graphs are slightly increase, the Java mode graphs are fixed at 2,600 MB. Because the JVM inspects the machine resources (size of memory and number of processors) and attempts to set various parameters to be optimal for long-running in the adaptive heap size mode. As for 30-sequence test data, the results are shown in figure 4. The native mode graphs greatly grow up and reach the maximum of virtual memory, however the Java mode graphs are the same as figure 3. Figure 5 shows the page faults/*sec* as function of elapsed time and the number of sequences are fixed at 20 sequences. The enormous page faults happen during the native modes are running, so that CPU wastes the times to manage with page faults. Also, figure 6 is similar to figure 5 for 30-sequence test data.

Table 1: Excelsior JET and JVM tuning parameters

Variables	Values
Garbage collection	2 threads
Maximum heap size	Adaptive size
Stack size	1 MB
Just-In-Time compiler	Enable

Table 2: Average execution times of MDR software on Dual-core Processor

#seq, #len	Execution Time (*sec*)	
	Native	JVM
20, 400	44.20	39.00
20, 600	58.00	50.40
20, 800	75.20	62.00
30, 400	236.80	205.60
30, 600	322.80	267.20
30, 800	444.40	338.20

Table 3: The running order of MDR modes and test data

Order	MDR mode	Sequence lengths
1^{st}	Native	400
2^{nd}	Java	400
3^{rd}	Native	600
4^{th}	Java	600
5^{th}	Native	800
6^{th}	Java	800

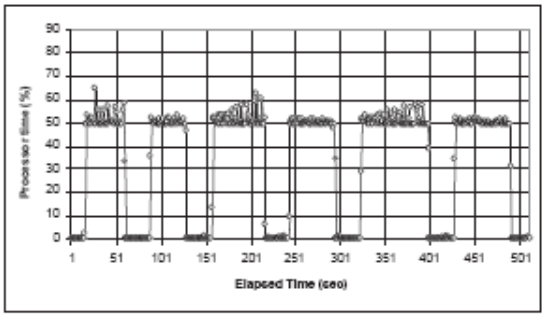

Figure 1: Percentage of processor time as function of elapsed time. The sequences lengths are varied but the number of sequences are fixed at 20 sequences.

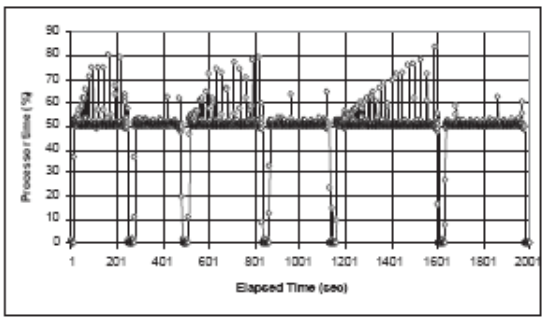

Figure 2: Percentage of processor time as function of elapsed time. The sequences lengths are varied but the number of sequences are fixed at 30 sequences.

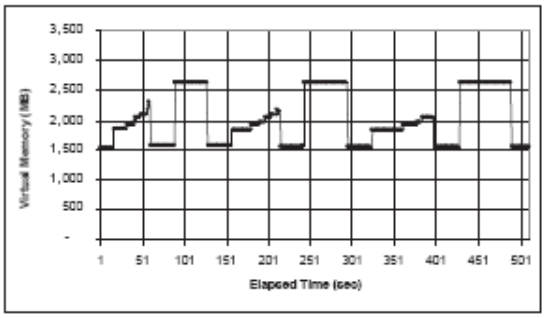

Figure 3: Virtual memory usage as function of elapsed time. The sequences lengths are varied but the number of sequences are fixed at 20 sequences.

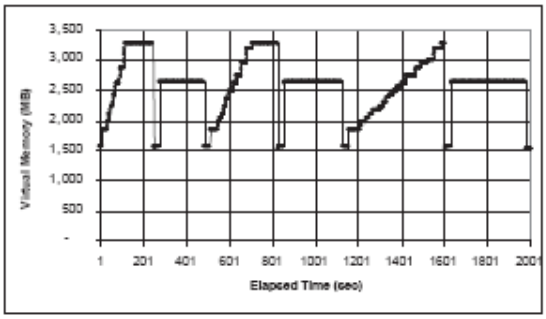

Figure 4: Virtual memory usage as function of elapsed time. The sequences lengths are varied but the number of sequences are fixed at 30 sequences.

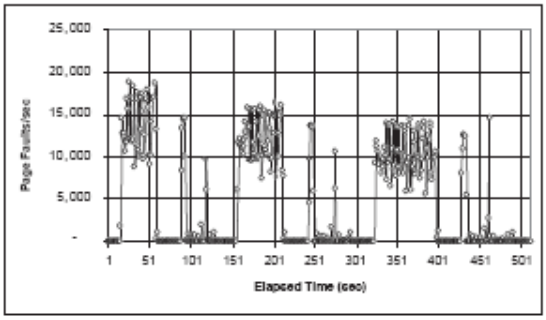

Figure 5: Page faults/*sec* as function of elapsed time. The sequences lengths are varied but the number of sequences are fixed at 20 sequences.

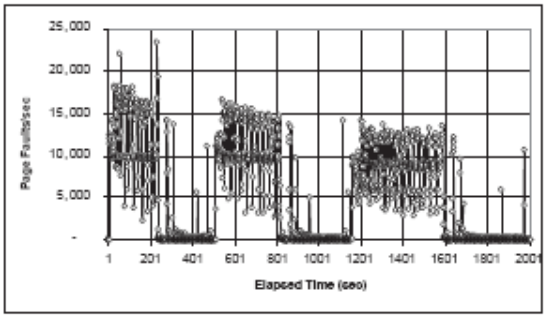

Figure 6: Page faults/*sec* as function of elapsed time. The sequences lengths are varied but the number of sequences are fixed at 30 sequences.

5. CONCLUSION

In this paper, we compared the performance of MDR software between Java and native modes. Through the heavy loads, we profiled and focused on execution time, memory usage and CPU usage. As for the current version of MDR software, the overall performance of Java mode is better than the native mode running in our environment. This paper can be helpful to tuning the Java system for higher throughput.

6. REFERENCES

[1] M. Mitchell and A. Samuel, "Gcc 3.0: The state of the source," *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

[2] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, and A. Yeryomin, "Overview of excelsior jet, a high performance alternative to java virtual machines," *Third International Workshop on Software and Performance*, pp. 104–113, 2002.

[3] C.A. Hsieh, J.C. Gyllenhaal, and W.W. Hwu, "Java bytecode to native code translation: The caffeine prototype and preliminary results," *Proceedings of the 29th Annual International Symposium on Microarchitecture*, 1996.

[4] S. Mintchev and V. Getov, "Automatic binding of native scientific libraries to java," in *ISCOPE*, 1997, pp. 129–136.

[5] M. Cabral de Araujo Gois and A.C. Frery, "An improved architecture for java domain transformation applications," *Proceedings of the XV Brazilian Symposium on Computer Graphics and Image Processing*, 2002.

[6] L.W. Hahn, M.D. Ritchie, and J.H. Moore, "Multifactor dimensionality reduction software for detecting genegene and geneenvironment interactions," *Bioinformatics, Oxford University Press*, vol. 19, no. 3, pp. 376–382, 2003.

[7] http://www.excelsior-usa.com, "Excelsior jet," .


29th Electrical Engineering Conference
Introduction
Organization
Important Dates
Call for Papers
Paper Submission
Technical Program
Exhibition
Registration
Accommodation
Sponsors
Links
General Information
Contact Us
ประกาศ
ตารางการนำเสนอบทความสำหรับ
การประชุมวิชาการทางวิศวกรรมไฟฟ้า ครั้งที่ 29 (EECON-29)
การลงทะเบียนระบบ Offline
วันสุดท้าย ของการลงทะเบียนล่วงหน้าของบุคคลทั่วไป นิสิต นักศึกษา
วันที่ 3 ต.ค. 2549
โปรดอ่าน...ด่วน !!!
ติดต่อจองห้องพักได้ที่นี่
สมาวิชาการวิศวกรรมไฟฟ้าแห่งประเทศไทย
http://www.eecon-thailand.org
EECON29
WEBBOARD
Copyright (c) 2006 Department of Electrical Engineering
Organized By
Sponsored By
Western Digital
NECTEC
EECON29
Important Dates
Full Paper Submission Due
July 10, 2006
Acceptance Notification
September 4, 2006
Manuscript Due
September 21, 2006
Conference Date
November 9-10, 2006

Optimized Multithreading Multiple Sequence Alignment for Intel Dual-core Processor

Kridsadakorn Chaichoompu and Surin Kittitornkun
Dept.of Computer Engineering
Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang
Bangkok 10520 ,Thailand
E-mail: {s7060809,kksurin}@kmitl.ac.th

Abstract

Sequence alignment of multiple nucleotides or amino acids is an important tool in bioinformatics. The most widely used tool for aligning multiple protein or nucleotide sequences called ClustalW, which consists of three stages: pairwise alignment, guide tree generation and progressive alignment. This paper proposes the methodology that assists the compiler to optimize ClustalW. Our goal is to maximize the throughput of execution on multithreading ClustalW called MT-ClustalW: our previous work. As a result, MT-ClustalW with optimization is able to fully utilize the machine resources and gains higher throughput on multicore and SMP (Symmetric MultiProcessor) machines. The experiment results show that our methodology can assist the compiler to optimize the code better than only compiler-optimization and achieve a better speedup over the sequential ClustalW and original MT-ClustalW.

Keywords: multiple sequence alignment, ClustalW, SIMD, parallel computing, dual-core processors

1 Introduction

Multiple sequence alignment is an important tool that identifies diagnostic patterns to characterize protein families. Many multiple sequence alignment tools have been proposed to reduce the high computation time of fully performing alignment of all sequences. ClustalW [1] particularly is the most popular sequential program for multiple sequence alignment, and CLUSTALX [2] is a graphical interface version of ClustalW.

Overview of parallel processing, a fast implementation of the Smith-Waterman sequence-alignment algorithm using Single-Instruction, Multiple-Data (SIMD) technology is presented. This implementation is based on the MultiMedia eXtensions (MMX) and Streaming SIMD Extensions (SSE) technology that is embedded in Intel's latest microprocessors. Six-fold speed-up relative to the fastest previously known Smith-Waterman implementation on the same hardware was achieved by an optimized 8-way parallel processing approach. [3] More about Smith-Waterman algorithm, a preliminary implementation of Smith-Waterman algorithm using a new chip multiprocessor architecture with multiple Digital Signal Processors (DSP) on a single chip leading to high performance at low cost. [4] Several methods of computing sequence alignments with limited memory per processing element on SIMD processing elements were presented in [5]. And this paper [6] improves alignment times by either reducing the alignment sensitivity or by developing specialized hardware. The solution comes in the form of parallel processing hardware such as Paracels GeneMatcher and Compugens Bioccelerator.

This work presents the methodology that assists the compiler to optimize the source code of ClustalW and MT-ClustalW and gain higher throughput. Our achievement is considerable and done by dual-core processor which is much less expensive workstation than the high-end multiprocessor.

2 ClustalW and Multithreading

2.1 Sequential ClustalW

ClustalW has become the most popular algorithm for multiple sequence alignment. This program implements a progressive method for multiple sequence alignment. As a progressive algorithm, ClustalW adds sequences one by one to the existing alignment to build a new alignment. The order of the sequences to be added to the new alignment is indicated by

a pre-computed phylogenetic tree, which is called a guide tree. The guide tree is constructed using the similarity of all possible pairs of sequences. The algorithm consists of 3 phases that are described follows: (1) Distance Matrix – All pairs of sequences are aligned separately in order to calculate a distance matrix based on the percentage of mismatches of each pair of sequences. (2) Neighbor joining – The guide tree is calculated from the distance matrix using a neighbor joining algorithm. (3) Progressive alignment – The sequences are progressively aligned following the guide tree.

2.2 MT-ClustalW

A fully multithreading version of ClustalW called MT-ClustalW [7] which significantly improves the performance of the traditional ClustalW. MT-ClustalW was modified from ClustalW-SMP and was done by using pthread library with MUTEX object as a synchronization object. In the proposed MT-ClustalW, all stages: the distance matrix, the guide tree and the progressive alignment; are divided into a number of threads and executed on a Pentium-D machine. So that bioinformatics laboratories are able to use this MT-ClustalW with much less energy consumption on multicore and SMP (Symmetric Multi-Processor) machines than that of PC clusters.

3 Improvement of MT-ClustalW (Enhanced version)

We used the Intel C++ Compiler for Windows to optimize the MT-ClustalW by enabling the /QxP option which optimizes code for Intel Core Duo processors and Intel Core Solo processors, Intel Pentium 4 processors with Streaming SIMD Extensions 3, and compatible Intel processors with Streaming SIMD Extensions 3 (SSE3). With the /QxP option, the compiler automatically traces the code and optimizes the loops. As we have done, only the simple loops were optimized. So we had to profile and change some codes to assist the compiler for optimizing the loops.

We used the Intel VTune Performance Analyzer to profile ClustalW in debug mode and look for the hotspots within the functions. After profiling, we focus on the top-usage functions to search for the hotspots and modify the codes with our guide methodologies.

Here are the methodologies that we used and applied to the codes for assisting the compiler to optimize the code better than original ClustalW 1.8. The methodologies are described as follows: Loop reversal, Loop fission, Type Casting, and Reduce procedure calls. The profiling result and the applied methodologies were shown in Table 1

Loop reversal — That is to run a loop backward. Reversal of `for` loops is always legal, since the execution is not defined in terms of the order of the index set. Thus, loop reversal is legal only when the loop carries no dependence relations. Here the example code, this loop is not vectorized by the compiler.

```
for (i=se2;i>0;i--) {
    HH[i] = -1;
    DD[i] = -1;
}
```

After we reverse the loop, it can be vectorized as follows.

```
for (i=1;i<=se2;i++) {
    HH[i] = -1;
    DD[i] = -1;
}
```

Loop fission — A single loop can be broken into two or more smaller loops. Loop fission can break up the block of conditionally executed statements. To apply loop fission, a temporary array must be used to hold the result of the variable which is not array. The temporary array is used to guard the execution of the statements after fission. The example code is shown as follow:

```
for (j=0;j<=N;j++){
    hh = HH[j] + RR[j];
    if (hh>=midh)
       if (HH[j]!=DD[j]&&RR[j]==SS[j]){
          midh=hh;
          midj=j;
       }
}
```

After applying loop fission, the code contains 2 loops. The first loop can be vectorized, but the second loop can not. However, we can make the compiler vectorize the second loop by using a temporary array to hold the result of the conditional test.

```
for (j=0;j<=N;j++){
    temp[j] = HH[j] + RR[j];
}
for (j=0;j<=N;j++) {
    if (temp[j]>=midh)
       if (HH[j]!=DD[j]&&RR[j]==SS[j]){
          midh=temp[j];
          midj=j;
       }
}
```

Type Casting — Converting an expression of a given type into another type. Type casting can be used to promote the variable from `integer` to `float`, this can help the compiler to vectorize the loop using the Intel MMX technology. The example is shown as follow:

```
for (ix=0;ix<=max_aa;ix++) {
    score+=(profile1[n][ix]
           *profile2[m][ix]);
}
```

After applying this method, the loop is vectorized. However, don't be forget to convert the result of the statement to the original type.

```
for (ix=0; ix<=max_aa; ix++) {
    score+=(int)((float)profile1[n][ix]
         * (float)profile2[m][ix]);
}
```

Procedure call reduction — Calling the procedure in the loop, the processor takes much overhead. The code can reduce the procedure calls by using `Macro`. The example code can be rewrite as `Macro` and is shown as follow:

```
static sint calc_score(sint iat,sint jat,
            sint v1,sint v2) {
   sint ipos,jpos,ret;
   ipos = v1 + iat;
   jpos = v2 + jat;
   ret=matrix[(int)seq_array[seq1][ipos]]
       [(int)seq_array[seq2][jpos]];
   return(ret);
}
```

After rewriting as `Macro`, the execution time reduces. The `Macro` code is like this:

```
#define calc_score(iat,jat,v1,v2)
   matrix[(int)seq_array[seq1][v1+iat]]
   [(int)seq_array[seq2][v2 + jat]]
```

4 Experiment Results

We implement our methodology using Intel C++ Compiler 9.0. The experiment was conducted using a 2.8GHz Intel 820 Pentium D (Dual-core) processor with 2 GB of memory. This computer runs MS Windows XP pro service pack 2 with no other applications installed. We compared the results of MT-ClustalW with ClustalW and got the same results. The elapsed times are measured as well as the wall clock time between the distance matrix stage and the progressive alignment stage. All measured times exclude the times that were taken to read the input data from a file and write the solution into a file. Furthermore, all measured times were measured when there is no one using the system except this experiment.

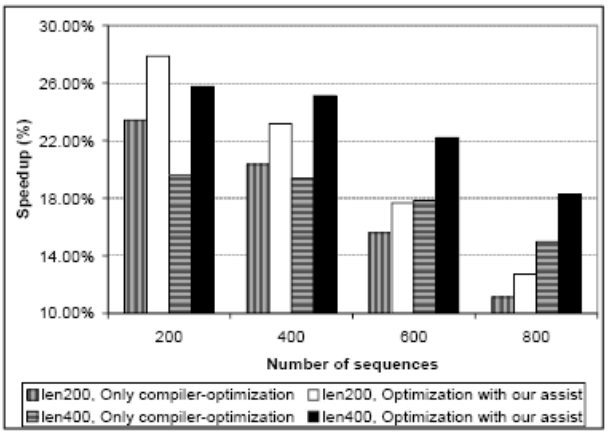

Figure 1: Speedup of the optimized versions of ClustalW as a function of number of sequences. The sequence lengths are fixed at 200 and 400 amino acids.

With the different running mode, Table 2 shows the elapsed times of ClustalW and MT-ClustalW in each stage. The elapsed times mainly reduce in the distance matrix stage for both ClustalW and MT-ClustalW, but little reduce in the neighbor Joining stage and the progressive Alignment stage. With the same test data set, Figure 1 shows the speedups of the optimized versions of ClustalW as a function of number of sequences. All speedups are compared with original ClustalW. It can be observed that ClustalW which is optimized with our assist gains significantly higher speedup than only compiler-optimization. Also when the sequence length changes from 200 to 400 amino acids, ClustalW which is optimized with our assist achieves higher speedup. However, the speedups of ClustalW of both versions decrease when the number of sequences is large. As same as ClustalW result, Figure 2 shows the speedups of the optimized versions of MT-ClustalW, but MT-ClustalW which is optimized with our assist gains much higher speedup when the sequence length is fixed at 400 amino acids. More about the results of MT-ClustalW, see the previous research in [7].

The proposed methodology fully utilizes the optimization feature in ClutalW and achieves the better speedup of ClustalW. MT-ClustalW is faster than ClustalW especially when the number of sequences is large. This will be compatible with the massive work for the alignment and gains more throughput.

Table 1: Profiling result and Optimization methodology of ClustalW

Function	Clockticks (%)	Optimization methodology*
diff	33.36	A, B
prfscore	15.93	C
forward_pass	14.91	-
calc_score	12.93	D
reverse_pass	11.45	A

*Note: A is Loop reversal, B is Loop fission, C is Type Casting, and D is Procedure call reduction

Table 2: Elapsed times of ClustalW and MT-ClustalW with 800 sequences and 400 amino acids.

Running mode*	Elapsed times (*ms*)		
	Distance Matrix	Neighbor Joining	Progressive Alignment
I	1,227,328	944,140	80,047
II	1,166,000	944,218	78,891
III	692,984	514,672	68,734
IV	147,813	515,828	30,906

*Note: Running mode defines as follows: (I) ClustalW with optimization (II) ClustalW with optimization and our assist (III) MT-ClustalW with optimization (IV) MT-ClustalW with optimization and our assist

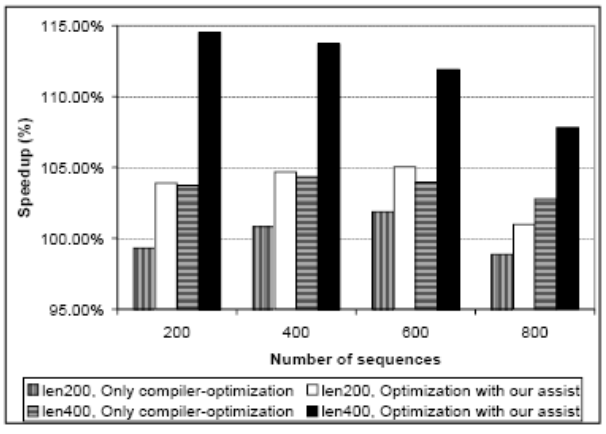

Figure 2: Speedup of the optimized versions of MT-ClustalW as a function of number of sequences. The sequence lengths are fixed at 200 and 400 amino acids.

5 Conclusion

In this paper, we presented a enhanced version of MT-ClustalW which significantly improves the performance of the traditional ClustalW. The proposed MT-ClustalW is optimized for a Pentium-D machine by using the Intel C++ compiler. The proposed methodology shows better speedups than the MT-ClustalW and much better than that of sequential ClustalW.

References

[1] J.D. Thompson, D.G. Higgins, and T.J. Gibson, "Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice," *Nucleic Acids Research 22*, vol. 22, no. 22, pp. 4673–4680, 1994.

[2] J. D. Thompson, T. J. Gibson, F. Plewniak, F. Jeanmougin, and D. G. Higgins, "The clustal_x windows interface: flexible strategies for multiple sequence alignment aided by quality analysis tools," *Nucleic Acids Research, 1997*, vol. 25, no. 24, pp. 48764882, 1997.

[3] T. Rognes and E. Seeberg, "Six-fold speed-up of smith-waterman sequence database searches using parallel processing on common microprocessors," *Bioinformatics 2000*, vol. 16, no. 8, pp. 699–706, 2000.

[4] X. Meng and V. Chaudhary, "Bio-sequence analysis with cradles 3soc software scalable system on chip," *ACM Symposium on Applied Computing*, 2004.

[5] J.A. Grice, R. Hughey, and D. Speck, "Parallel sequence alignment in limited space," *Proceedings of International Conference Intelligent Systems for Molecular*, pp. 145–153, 1995.

[6] F. Lau, "An integrated approach to fast, sensitive, and cost-effective smith-waterman multiple sequence alignment," *Bioinformatics Module*, 2000.

[7] K. Chaichoompu, S. Kittitornkun, and S. Tongsima, "Mt-clustalw: Multithreading multiple sequence alignment," *Proceedings of the IEEE IPDPS 2006 (HiCOMB06)*, 2006.

About the authors

Kridsadakorn Chaichoompu is master student. Interesting in Bioinformatics, Parallel computing, High performance computing and Multithreading algorithm. Previous work is MT-ClustalW : Multithreading multiple sequence alignment.

Author Biography

Personal Data	
• Name	Kridsadakorn Chaichoompu, Mr.
• Date of Birth	17 Aug. 1981 (Thailand)
• Nationality	Thai
• Address	Kamon Apartment, 556/1 Moo 1, Soi Rimsuan, Ladkrabang, Bangkok, 10520, Thailand
• Mobile	+66 (0)8 1207 1918
• E-mail	kridsadakorn_chai@hotmail.com
Education	
• Jun.2004-Oct.2006	King Mongkut's Institute of Technology, Ladkrabang (KMITL), Bangkok, Thailand ➔ Master of Engineering in Computer Engineering *(Research plan)*
• Jun.2000-Apr.2004	➔ Bachelor of Engineering in Computer Engineering *(GPA 3.33, 2nd Class Honor)*
Qualification	
• Language	Thai and English
• Programming Language	Assembly, Pascal, C/C++, Visual C++, Visual Basic, Java, JavaServlet, HTML/XHTML, ASP, JSP, PHP, JavaScript, VBScript
• Operating System	Ms Windows, Linux/Unix
• Networking/Web	TCP/UDP Networking, Apache HTTP Server, IIS
• Database System	SQL, Relational Database Design, MySQL
• Hardware	VDHL, Digital Circuit Design, MCS-8051
Research Interests	
	Bioinformatics, Intelligent system, Mobile & wireless computing, Parallel computing, Multithreading algorithm, and High performance computing.