UNIVERSITY OF NAIROBI

COLLEGE OF BIOLOGIAL AND PHYSICAL SCIENCES

SCHOOL OF COMPUTING AND INFORMATICS

BIO-SEQUENCE SEARCH ENGINE BASED ON DAMERAU LEVENSHTEIN DISTANCE ALGORITHM

BY

KENEI, K JONAH

P56/9080/2006

SUPERVISOR: EVANS MIRITI

SEPTEMBER 2011

Submitted in partial fulfillment of the requirements of the Master of Science in Computer Science

DECLARATION

I hereby declare that this thesis is based on the results found by me. Materials of work found by other researchers are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Sign: [illegible] Date: 03/10/2011

Jonah Kenei
P58/9080/2006

This project has been submitted as partial fulfillment of the requirements for the Master of Science degree in Computer Science of the University of Nairobi with my approval as the University supervisor.

Sign: [illegible] Date: 04/10/2011

EVANS MIRITI

ABSTRACT

Bio-sequence databases currently require enormous computing power to processes the vast quantities of data available. Further, DNA sequencing projects are generating data at an exponential rate; thus, new, faster methods and techniques of processing this data are needed. One frequently used type of processing involves searching a sequence database for similar sequences. In this project we present a parallelized Damerau Levenshtein distance algorithm using multi-threading programming. This algorithm efficiently distributes the patterns to be searched on multiple threads to achieve rapid sequence matching operation. The algorithm is designed to fully exploit thread level parallelism to enhance searching speed. By distributing a large number of patterns over multiple threads, the algorithm shows better performance. From detailed experiments and performance analysis, our algorithm shows remarkable performance gain compared to the original Damerau Levenhstein algorithm. 36% average performance gained is realized.

ACKNOWLEDGEMENTS

I would like to express my heartfelt appreciation to Evans Miriti for being my advisor and mentor. His motivation, suggestions and insights for this project have been invaluable. Without his support and guidance this project would not have been possible.

I would also like to extend my gratitude to my other evaluation panel members, Dr. Wanjiku Ng'anga and Robert Oboko, for their time and input provided during the work.

Last, but definitely not the least, I would like to thank my wife Juliet for her encouragement and help, and for bearing with me for the whole year while I worked on the project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYNMS

DNA – Deoxyribonucleic acid, Molecule that encodes genetic information

RNA - Ribonucleic acid, Molecule that plays a role in transferring information from DNA to protein forming system of the cell

BLAST- Basic Local Alignment Search Tool

BLOSUM- Blocks Substitution Matrix

PAM - Percent accepted mutation

HMM – Hidden Markov Model

DFA- Deterministic Finite Automaton

NFA- Non Deterministic Finite Automaton

CHAPTER 1: INTRODUCTION

1.1 Introduction

Sequence searching remains a computational problem as the total number of sequences in the underlying databases grows exponentially with the progress of research. Searching large biological sequence databases calls for the need for highly efficient algorithms. DNA sequences are generally very long chains of sequentially linked nucleotides (Gusueld D 1973). Algorithms devised for the comparison of molecular sequences are based on the concept of string matching. DNA sequences which hold the code of life for every living organism can be abstractly viewed as very long strings over a four letter alphabet of A, C, G and T (Fahim Sufi 2005).

The field of bioinformatics aims to develop techniques which enable the analysis of bio-sequences (DNA, RNA and protein sequences). The DNA sequences make up the genome of living organisms and are massively long strings of four different nucleotides: Adenine, Cytosine, Guanine, and Thymine. A frequent challenge given DNA sequences is motif detection, i.e., the detection of patterns of nucleotides that are of some biological significance. Research on motif detection has benefited the development of many exact as well as approximate string search algorithms as well as more elaborate techniques for inferring second-level properties (Richard Durbin et al 1998).

The analysis of nucleic acid and protein sequences is one of the oldest problems in computational biology. The comparison of biological sequences has become a fundamental aspect of molecular biology and its success as a scientific tool has transformed the science of Biology. Sequence comparison has proven to be a powerful means to infer evolutionary and functional relationships. It has enabled biologists to find families of related genes and proteins and to clarify the workings evolution at the molecular level.

Biologists use sequence comparison algorithms to:

i. Measure the degree of difference or similarity between sequences
ii. Construct optimal alignments
iii. Search for similar sequences in databases.

Large databases of sequence data have been compiled and it is standard practice to compare and submit newly discovered sequences to the databases. The human genome and the genomes of hundreds of (mostly microbial) model organisms have been completely sequenced. The Genbank nucleic acid database at the NCBI held roughly 45 billion base pairs as of 2004. The human genome alone is 3 billion base pairs long (J. Christopher Bare 2005).

Nucleic acids (DNA and RNA) are the molecular carriers of hereditary information in living organisms. These molecules are polymers of four nucleotides. DNA sequences are usually represented as strings of characters over the alphabet {A, C, G, T}, where each character corresponds to a nucleotide base. Similarly, RNA sequences are represented as strings over the alphabet {A, C, G, U}. The size of the DNA or RNA of an organism (its genome size) varies significantly. For example, the size of viral genomes ranges from few thousand to around million nucleotide base pairs; bacterial genomes sizes range from hundreds of thousands to less than ten million base pairs, and the size of mammalian genomes ranges between one and eight billion base pairs (Gregory 2005), e.g. a human genome comprises around 3 billion base pairs (Mirjana Domazet-Lošo 2010).

This project aims to first study and review various sequence matching algorithms and methods used in sequence comparisons. Then from this review, proposed the use of a bio-sequence search engine based on Damerau Levenshtein distance algorithm for bio-sequence analysis. This algorithm best models biological mutations (insertion, deletion, substitution and transposition). The dissertation will demonstrate through experimentation and graphical representation, of results obtained, how this search engine can be applied to perform biological sequence matching.

The general goal of this thesis is to review various sequence comparison algorithms and propose a multithreaded algorithm based on edit distance metric (Damerau Levenshstein distance algorithm) and thus contribute to the area of sequence comparison algorithms and similarity search by improvements of the contemporary methods and by proposal of new methods to analyze bioinformatics data.

The schematic for every living organism is stored in long molecules known as chromosomes made of a substance known as DNA (deoxyribonucleic acid). Each cell in an organism has a complete copy of its DNA, also known as its genome which is conveniently modeled as a sequence of symbols (alternately referred to as nucleotides or bases) in the DNA alphabet {A, C, T, G}. In humans, and most mammals, this sequence is about 3 billion bases long (Archie Russell & Jason Hogg 2001).

Common to all life is the flow of information within a cell from DNA to RNA to protein, first established in the 1950's by Francis Crick. DNA serves as the storage repository of genetic information. Genes are transcribed to mRNA, which carries the information to ribosomes. A ribosome is a molecular factory that constructs proteins using mRNA as a template in a process called translation. Nucleic acids, DNA and RNA, along with proteins are the central molecules of biology. Nucleic acids consist of long chains of chemical subunits called nucleotides or bases. Similarly, a protein is a long chain of amino acids. In either case, the sequence of a molecule defines the exact order of the subunits along the length of the chain. Biological sequences can be represented as strings over an alphabet that contains one letter for each type of chemical subunit (J. Christopher Bare 2005).

To understand the biological side of bioinformatics, one must understand basic molecular biology. Three molecule types are very important: deoxyribonucleic acid (DNA), ribonucleic acid (RNA) and proteins. DNA, RNA and proteins make up the genetic code and the products of the genetic code that direct the processes of life (Laurie Pepper 2001).

Molecule	Alphabet
DNA	{A,T,G,C}
RNA	{A,U,G,C}
Protein	{A,C,D,E,F,G,H,I,K,L,M,N,P,Q,R,S,T,V,W,Y}

Table 1.1 Bio-sequences alphabet

A three letter chunk of nucleic acid, called a codon, translates into one amino acid. The mapping is defined by the genetic code. A gene is said to code for a protein if the sequence of the gene translates into the sequence of the protein. With few exceptions, the genetic code is universal (J. Christopher Bare 2005).

DNA (Deoxyribonucleic acid) sequences hold the code of life of living organisms. Approximate string matching on DNA sequences is very important in research in the fields of medicine and health. Although the approximate string matching problem has been studied extensively in the field of computer science, many solutions cannot be applied directly on DNA data. This is because DNA data is a very large data set (GenBank had recorded more than 20Gbp of DNA sequences at June 2002) (Sudha Surendirath 2005).

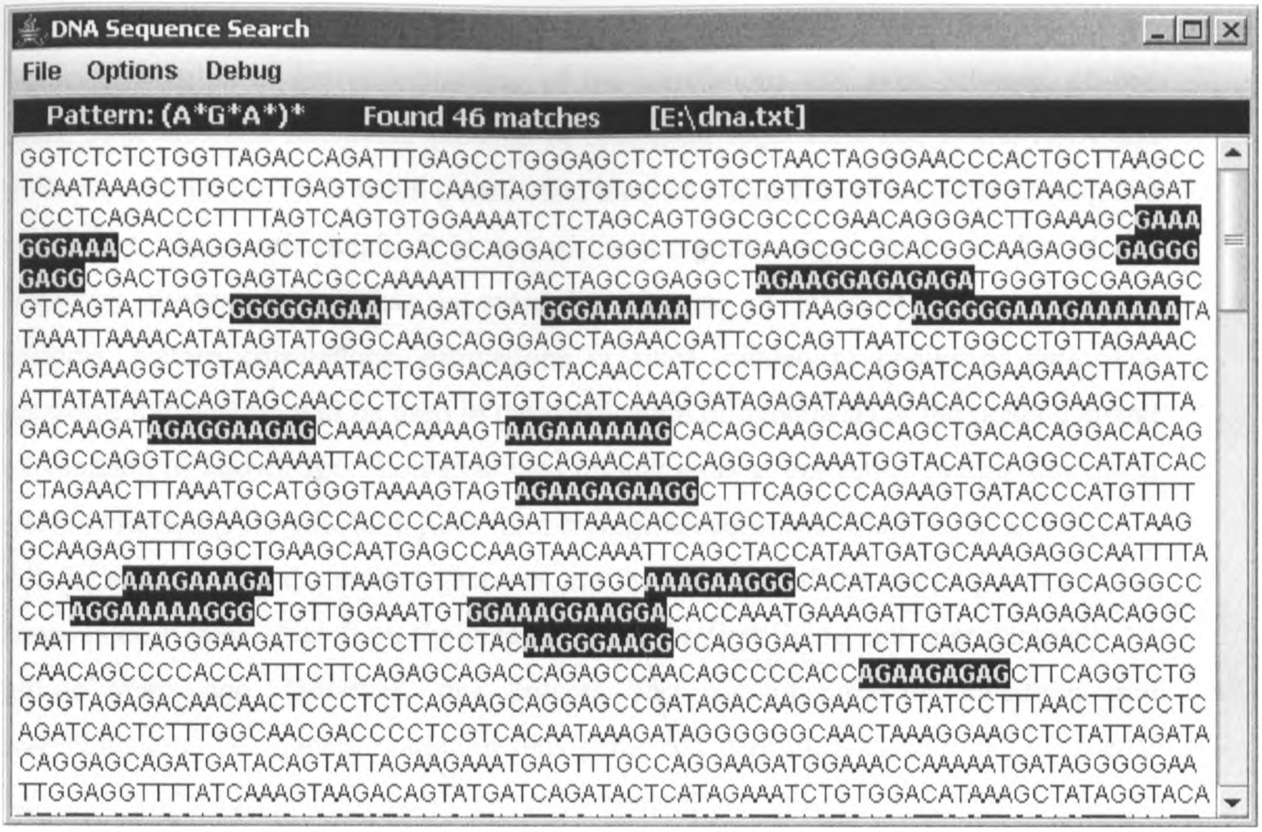
New DNA sequencers produce massive amounts of short reads of DNA text in a single run. If a reference genome is known, a first step in processing these short reads is to map them to the reference genome. As the number of these short sequences is very large, new efficient multiple string matching algorithms are needed to complete this task.

Initially, efficient algorithms were designed to perform simple searches on strings, as might be required in a word processor. However, in the age of large scale DNA sequencing the desire for, and variety of, string manipulation algorithms has increased dramatically (Xia Cao et al 2005).

The genetic material of organisms evolves by discrete mutations, most prominently substitutions, insertions and deletions of nucleotides. Since the genetic material is stored in DNA sequences and reflected in RNA and protein sequences, it makes sense to compare two or more biological sequences to look for similarities and differences that can be used to infer the relatedness of the sequences (Christian P. Kreibich 2007).

The process of extracting the sequence of a DNA molecule is called “DNA sequencing”. Extracting the sequence of a single member of a species in order to gain a more complete understanding of its biology is however not the only application of DNA sequencing. Differences in individual members of a population are also of interest. Mutations in individuals are of scientific interest where they can, for example, be used to track the evolution of a population. Mutations can also cause or increase susceptibility to disease or affect the drug resistance of an individual. Because of the utility of DNA sequencing in biomedical applications it has become of huge economic importance.

A DNA sequence is created by the DNA sequencing process. A byproduct of this process is a long string containing a succession of letters A, C, G, and T, representing the four nucleotide subunits (a chemical compound) of a DNA strand - **A**denine, **C**ytosine, **G**uanine, and **T**hymine bases. A sample string produced this way resembles a pattern similar to AAAGTCTGAC but often longer in length. A DNA sequence file contains one complete sequence – a sample is shown in Figure 1.1 (Kuha Mahalingam and Omar Bagasra 2008).



```
Pattern: (A*G*A*)*      Found 46 matches      [E:\dna.txt]
GGTCTCTCTGGTTAGACCAGATTTGAGCCTGGGAGCTCTCTGGCTAACTAGGGAACCCACTGCTTAAGCC
TCAATAAAGCTTGCCTTGAGTGCTTCAAGTAGTGTGTGCCCGTCTGTTGTGTGACTCTGGTAACTAGAGAT
CCCTCAGACCCTTTTAGTCAGTGTGGAAAATCTCTAGCAGTGGCGCCCGAACAGGGACTTGAAAGCGAAA
GGGAAACCAGAGGAGCTCTCTCGACGCAGGACTCGGCTTGCTGAAGCGCGCACGGCAAGAGGCGAGGG
GAGGCGACTGGTGAGTACGCCAAAAATTTTGACTAGCGGAGGCTAGAAGGAGAGAGATGGGTGCGAGAGC
GTCAGTATTAAGCGGGGGAGAATTAGATCGATGGGAAAAAATTCGGTTAAGGCCAGGGGGAAAGAAAAAATA
TAAATTAAAACATATAGTATGGGCAAGCAGGGAGCTAGAACGATTCGCAGTTAATCCTGGCCTGTTAGAAAC
ATCAGAAGGCTGTAGACAAATACTGGGACAGCTACAACCATCCCTTCAGACAGGATCAGAAGAACTTAGATC
ATTATATAATACAGTAGCAACCCTCTATTGTGTGCATCAAAGGATAGAGATAAAAGACACCAAGGAAGCTTTA
GACAAGATAGAGGAAGAGCAAAACAAAAGTAAGAAAAAAGCACAGCAAGCAGCAGCTGACACAGGACACAG
CAGCCAGGTCAGCCAAAATTACCCTATAGTGCAGAACATCCAGGGGCAAATGGTACATCAGGCCATATCAC
CTAGAACTTTAAATGCATGGGTAAAAGTAGTAGAAGAGAAGGCTTTCAGCCCAGAAGTGATACCCATGTTTT
CAGCATTATCAGAAGGAGCCACCCCACAAGATTTAAACACCATGCTAAACACAGTGGGCCCGGCCATAAG
GCAAGAGTTTTGGCTGAAGCAATGAGCCAAGTAACAAATTCAGCTACCATAATGATGCAAAGAGGCAATTTTA
GGAACCAAAGAAAGATTGTTAAGTGTTTCAATTGTGGCAAAGAAGGGCACATAGCCAGAAATTGCAGGGCC
CCTAGGAAAAAGGGCTGTTGGAAATGTGGAAAGGAAGGACACCAAATGAAAGATTGTACTGAGAGACAGGC
TAATTTTTTAGGGAAGATCTGGCCTTCCTACAAGGGAAGGCCAGGGAATTTTCTTCAGAGCAGACCAGAGC
CAACAGCCCCACCATTTCTTCAGAGCAGACCAGAGCCAACAGCCCCACCAGAAGAGAGCTTCAGGTCTG
GGGTAGAGACAACAACTCCCTCTCAGAAGCAGGAGCCGATAGACAAGGAACTGTATCCTTTAACTTCCCTC
AGATCACTCTTTGGCAACGACCCCTCGTCACAATAAAGATAGGGGGGCAACTAAAGGAAGCTCTATTAGATA
CAGGAGCAGATGATACAGTATTAGAAGAAATGAGTTTGCCAGGAAGATGGAAACCAAAAATGATAGGGGGAA
TTGGAGGTTTTATCAAAGTAAGACAGTATGATCAGATACTCATAGAAATCTGTGGACATAAAGCTATAGGTACA
```

Figure 1.1 DMST graphical interface with highlighted matches

With the development of Molecular Biology during the last decades, we are witnessing an exponential growth of both the volume and the complexity of biological data. The Human Genome Project is providing the sequence of the 3 billion DNA bases that constitute the human genome. And, consequently, we are provided too with the sequences of about 100,000 proteins. Therefore, we are entering the post-genomic era. After having focused so much efforts on the accumulation of data, we have now to focus as much efforts, and even more, on the analysis of these data. This will enable us to learn more about gene expression, protein interactions and other biological mechanisms. Analyzing this huge volume of data is a challenging task because, not only, of its complexity and its multiple numerous correlated factors, but also, because of the continuous evolution of our understanding of the biological mechanisms. Classical approaches of biological data analysis are no longer efficient and produce only a very limited amount of information, compared to the numerous and complex biological

mechanisms under study. Actually, these approaches use only a very limited number of parameters, to represent the so-many correlated factors involved in the biological mechanisms. From here comes the necessity to use computer tools and develop new in silico high performance approaches, to support us in the analysis of biological data. And, hence, to help us in our understanding of the correlations that exist between, on one hand, structures and functional patterns of biological sequences, i.e., DNA, RNA and proteins, and, on the other hand, genetic and biochemical mechanisms. Data mining is a response to these new trends. It is one of the pre-processing steps in the knowledge discovery process. It consists in extracting nuggets of information, i.e., pertinent patterns, pattern correlations, estimations or rules, hidden in bodies of data. The extracted information will be used in the verification of hypothesis or the prediction and explanation of knowledge. Biological data mining aims at extracting motifs, functional sites or clustering/classification rules from biological sequences (Ela Hunt et al 2001)
All the existing and the extinct genomes are the outcome of the copying process that happens each generation from the emergence of the first living cell approximately 3.8 billion years ago. However, this process was accompanied by mutations and recombination. The genetic variation thus generated permitted adaptation to different habitats, which resulted in the diversity of present and extinct organisms. Thus, the evolution of organisms or sequences can be envisaged as a branching process where every pair of organisms or sequences has a common ancestor at a varying depth of an emerging tree (Mirjana Domazet-Lošo 2010)
The field of bioinformatics uses computers and algorithms to solve problems from Biology. This is useful, because in biology scientists often need to analyze large amounts of data. One example is the analysis of DNA using computers. We can use them to compare DNA strings, and find how similar they are, which gives us a measure of how closely related two species are. Evolution is based on mutations of DNA strings. Atomic mutation is either an insertion, deletion or substitution of one of the base pairs that makes up the DNA string, denoted by C, G, A and T. We can look at two strings and find out what the least number of mutations are to go from one string to the other. This notion in general is called the edit distance between two strings. It measures how many atomic edits (insertions, deletions, substitutions) one needs to go from one string to the other.

1.3 Problem Statement

One of the most important tasks in bioinformatics is the search for similar genetic sequences in the data banks. With the new sequencing technologies, the size of these genetics data banks are growing exponentially (Dennis A. Benson et al 2007), and consequently the search time is growing too.

One of the widely used approaches has been sequence alignment algorithms. Pair wise alignment compares two pairs of sequences at a time, thus making practically impossible for large genomic databases. Multiple sequence alignment is NP complete in general and therefore not likely to be solvable in polynomial time (Michael R. Garey and David S. Johnson 1979). So it's very difficult to improve the speed of these methods greatly, especially in this DNA information explosion period. Much of the research worked done before have focused on sequence alignment algorithms which can be group into two approaches:

1. Dynamic programming such as Needleman-Wunsch or Smith-Waterman
2. Heuristic methods such as BLAST and FASTA

Dynamic programming methods are guaranteed to find all optimal alignments, but are relatively slow; heuristic methods are faster but less precise (M.A. Kentie 2005). Heuristic methods like BLAST, FASTA etc rely on heuristics and they fail on queries which do not have very similar sequences deposited in the database. Their speed is achieved by compromising the sensitivity of these algorithms (Clare Sansom 2000).

Another problem of sequence alignment algorithms based methods is the drawback of the increasing computational complexity with the increase in the number of the sequences as well as the size of the sequences (Susana Vinga and Jonas Almeida 2003).

Therefore the research into alignment free sequence analysis is necessary to overcome critical limitations of sequence analysis by alignment. In (Susana Vinga & Jonas Almeida 2003) it has been mentioned that although the pace of work in this area is increasing, the total number of published reports proposing or using alignment-free metrics is still slow.

This project in an attempt to solve the above problem tries to review and study various sequence comparison algorithms. Then from this review, propose a bio-sequence search engine based on alignment free sequence analysis algorithm. In the course of this study, we developed a new search engine based on Damerau Levehstein distance algorithm as an alternative to sequence alignment algorithms.

1.4 Purpose of the project

The purpose of this study to implement a parallelized version of Damerau Levenhstein distance algorithm to find effective ways of searching Biological sequence databases, searching for occurrences of finite sequences in a given biological database. Several sequence comparison algorithms were studied: string matching algorithms, sequence alignment and edit distance metric algorithms.

This project aims to study sequence matching algorithms and develop a bio-sequence search engine model using parallelized version of Damerau Levenshtein distance algorithm that is efficient i.e. provides optimal execution time

1.5 Objectives

1. To improve the efficiency of sequence comparison algorithms by using multi-threading programming model.
2. To highlight the challenges of bio-sequence similarity search and why traditional string searching algorithms are inadequate for string comparisons in Bioinformatics.
3. To implement a search engine model based on parallelized Damerau Levenshtein distance algorithm and demonstrate how it can be used in sequence comparisons.
4. To prove using experimentation and show how search engines can be used in sequence similarity search and can achieve both high performance and scalability.

1.6 Research Questions

1. Why are pattern matching algorithms important in Bioinformatics?
2. What are the various algorithms used for sequence similarity search?
3. What are the deficiencies of current sequence alignment algorithms used in searching large bio-sequence datasets?
4. Is it possible to improve the performance of Damerau Levenshtein distance algorithm by using software parallelism using multi-threading.programming?
5. What is the performance of Damerau Levenshtein distance algorithm under different number of threads?

1.7 Significant of the study

The research outcomes of this project will help Biologists and researchers in analyzing bio-sequences data such as DNA sequences. A search engine which is capable of searching and highlighting sequences with high sequence similarity will help Biologists infer the functional or structural similarity of biological sequences as well as providing direct evidence for the evolutionary relationships of organisms.

From the literature, most past research in sequence comparison have focused on sequence alignment algorithms, thus the proposed search engine based on parallelized Damerau Levenhstein distance algorithm will usher in interesting insights in research towards sequence comparison algorithms.

1.9 Organization of the chapters

The rest of the paper is organized as follows. In chapter 2 we provide the background and literature review of sequence comparison algorithms. Chapter 3 describes the research design, methodology and tools be used to solve the problem by this study. Chapter 4 provides a design and implementation of Damerau Levenshtein distance algorithm. Chapter 5 presents results analysis and discussion. Chapter 6 concludes the findings and outlines possible directions for future research.

CHAPTER 2: LITERATURE REVIEW

We present in summary form the most significant works done in the past, in areas that are related to our work. A large number of algorithms for performing bio-sequence comparison mostly employ sequence alignment algorithms. To speed up the operation, many researchers have also proposed parallel hardware approaches to the sequence matching algorithms. There has been little research on using edit distance metric algorithms, which is what we have employed in our design. On similar lines, we have proposed a novel Damerau Levenhstein distance algorithm solution.

2.1 Background Theory

The schematic for every living organism is stored in long molecules known as chromosomes made of a substance known as DNA (deoxyribonucleic acid). Each cell in an organism has a complete copy of its DNA, also known as its genome which is conveniently modeled as a sequence of symbols (alternately referred to as nucleotides or bases) in the DNA alphabet {A, C, T, G}. In humans, and most mammals, this sequence is about 3 billion bases long. Through a process known as protein synthesis, instructions in our DNA, known as genes, are interpreted by machinery in our bodies and transformed first into intermediate molecules called RNA, and then into structures called proteins, which are the primary actors in living systems. These molecules can also be modeled as sequences of symbols (Archie Russell & Jason Hogg 2001).

One of the most common computational operations on sequences is a sequence similarity search. A typical search compares one query sequence to a larger database of sequences and tries to find alignments between the query and database that reflect similarities between the two. These searches are valuable to researchers because of the nature of evolution, and the nature of scientific research. Researchers may be able to determine a gene in a mouse which is responsible for a condition such as obesity; a sequence similarity search helps pinpoint the analogues of that gene in humans. Similarity searches are also used within a single species; portions of a gene, called domains, are often duplicated among several genes. Finally, similarity searches allow researchers to infer which parts of our DNA are important. Relevant portions of our genome are less likely to have changed over time than less-important regions. A similarity search that detects regions in common between evolutionarily distant species has probably also detected very important genomic regions.

In biological sequences (DNA, RNA, or protein sequences), high sequence similarity usually implies significant functional or structural similarity. Understanding the relationship of a query DNA or protein genomic sequence to the known sequences in genomic databases allows molecular biologists to assign functions to poorly understood sequences. Therefore, similarity search in genomic databases is an important function in genome research as it is useful for discovering the location of functional sites, searching novel repeats and conducting comparative analysis of different genomic sequences. To cater for evolutionary mutations in genomic sequences and noise in the sequence data, approximate sequence matching is preferred to exact matching from the biologists' point of view when similarity search in genomic databases is conducted (Xia Cao 2006). We first review the different classes of sequence comparison algorithms namely, string matching algorithms, sequence alignment and then introduce the concept of edit distance as applied to string matching.

2.2 Searching Sequence Databases

There are a number of bio-sequence databases in different countries and publicly available services provide a major comprehensive collection of nucleotide data from repositories worldwide:

1. The GenBank nucleotide database that is maintained by the National Center for Biotechnology USA
2. National Center for Biotechnology Information (NCBI) in the USA (Benson et al 2005); and
3. The DNA Database of Japan (DDBJ) that is maintained by the National Institute of Genetics in Japan (Miyazaki et al 2004)
4. The European Molecular Biology Laboratory (EMBL) that is based in the United Kingdom

These bio-sequence database collections are regularly updated and mirror the contents of one another daily, so that recently deposited sequences in one database will soon appear in all three collections (Rapp & Wheeler 2005).

The GenBank nucleotide database currently contains more than 53 billion nucleotide base pairs stored in almost 50 million sequences from over 150,000 different organisms. The collection has roughly doubled in size every 1.4 years, which is faster than improvements in the processing power of modern workstations (Attwood & Higgs 2004). Further, this

trend is expected to continue with new technologies for high-throughput sequencing and support from scientific journals, many of which now require new sequences to be submitted to GenBank before the related work is published (Rapp & Wheeler 2005).
A common task in bioinformatics is the search of a database for the sequences with significant similarity to a query sequence. These search methods must balance two parameters: the sensitivity of the algorithm to the differences between sequences; and the time needed to search the entire database. These methods also incorporate a number of parameters so that a search can be tailored to the specific needs of the user.
Once a DNA sequence that codes for some protein with unknown function has been sequenced, the databases of DNA sequences are searched for similar DNA sequences. The function of the unknown piece of DNA may be inferred from DNA sequences with similar structure that are found in the databases. In many situations, knowing that two sequences are similar is enough, but in other situations such as constructing a phylogenetic tree, a one to one alignment between the bases of the sequences is required. The processes of evaluating the similarity of sequences and constructing an alignment between two sequences both make heavy use of algorithms to do approximate string matching. The size of these problems varies greatly, from aligning two short sequences to doing a complete n-wise comparison of all sequences in the GenBank database.
Sequence database searches cover a wide area of applications biologists use for their daily work where large databases need to be searched for sequences they have gained from experiments. To illustrate the process, we can consider the sequences for now as DNA sequences, i.e. strings over the alphabet ∑DNA = {A, C, G, T}.
Finding similar sequences helps biologists in many ways. First, it helps to determine, whether the sequences they are investigating have been identified and analyzed before. But even if a gene has not been identified before, finding similar genes helps to determine its function in the organism, because often similar genes code for proteins with similar functions. If we find a sequence in a new species that is similar to the sequence of hemoglobin in the human body for example, it is likely that this new sequence codes for proteins that are responsible for oxygen–transport in the red cells of the blood as well(Dominic Battr´ 2005).

2.3 String matching algorithms

Most sequence matching algorithms focus on sequence pattern itself without using the preprocessing phase. Other algorithms compare the pattern and the text from left to the right. Some other algorithms perform comparison from right to left. The performance of the algorithm depends upon the order in which the comparison is done. Several pattern matching algorithms have been developed to minimize the number of comparisons. Brute-force algorithm compares the pattern with the text from left to right. After each attempt it shifts the pattern by exactly one position to right. The time complexity is O(mn) in worst case and the number of text character comparison is 2n.Boyer-Moore algorithm (Boyer-Moore 1977) improves the performance by preprocessing the pattern by using two shift functions. The bad character shift and the good suffix shift. During the searching phase the pattern is aligned with the text and it is scanned from right to left. If a mismatch occurs the algorithm shifts the pattern with the maximum value taken between the two shift functions. The worst case complexity is O(mn).

In (Raju-Somayajulu 2010) the elements in the given patterns are matched one by one in the forward and backward until a mismatch occurs or a complete pattern matches. In the (Ziad A.A et al 2007) technique the algorithm scans the input file to find all occurrences of the pattern based upon the skip technique. Index is used as the starting point of matching; it compares the file contents from the defined point with the pattern contents, and finds the skip value depending upon the match numbers (ranges 1 to m-1) (Harspool 1980) does not use the good suffix function, instead it uses the bad character shift with right most character .The time complexity of the algorithm is O(mn). (Berry-Ravindran 1999) calculates the shift value based on the bad character shift for two consecutive text characters in the text immediately to the right of the window. This will reduce the number of comparisons in the searching phase. The time complexity of the algorithm is O(nm). (Sunday,1990) designed an algorithm quick search which scans the character of the window in any order and computes its shift with the occurrence shift of the character T immediately after the right end of the window. In (Raju-Somayajulu 2010) algorithm we first choose the value of k (a fixed value), and divide both the string and pattern into number of substring of length k, each substring is called a partition. If k value is 3 we call it as 3-partition else if it is 4 then it is 4-partition algorithm. We compare all the first characters of all the partitions, if all the characters are matching while we are searching then we go for the second character match and the process continues till the mismatch occurs or total pattern is matched with the sequence. If all the characters match then the pattern occurs in the sequence and prints the starting index of the pattern or if any character mismatches then we will stop searching and then go to the next index stored in the index table of the same row which corresponds to the first character of the pattern P. The Knuth-Morris-Pratt algorithm (Knuth-Morris-Pratt 1977) is based on the finite state machine automaton. The pattern P is pre-processed to create a finite state machine M that accepts the transition. The finite state machine is usually represented as the transition table. The complexity of the algorithm for the average and the worst case performance is O(m+n). In approximate pattern matching method the oldest and most commonly used approach is dynamic programming. By using dynamic programming approach especially in DNA sequencing (Needleman-Wunsch 1970) algorithm and (Smith-waterman 1981) algorithms are more complex in finding exact pattern matching algorithm. By this method the worst case complexity is O(mn). The major advantage of this method is flexibility in adapting to different edit distance functions. The first bit-parallel method is known as "shift-or" which searches a pattern in a text by parallelizing operation of non deterministic finite automaton. This automaton has m+1 states and can be simulated in its non deterministic form in O(mn) time. (Ukkonen 1985) proposed automaton method for finding approximate patterns in strings. He proposed the idea using a DFA for solving the inexact matching problem. Though automata approach doesn't offer time advantage over (Boyer-Moore 1997) algorithm for exact pattern matching. The complexity of this algorithm in worst and average case is O(m+n).In this algorithm, every row denotes number of errors and column represents matching a pattern prefix. Deterministic automata approach exhibits O(n) worst case time complexity. The main difficulty with this approach is construction of the DFA from NFA which takes exponential time and space. The (Knuth-Morris-Pratt 1977) algorithm is based on the finite state machine automaton. The pattern P is pre-processed to create a finite state machine M that accepts the transition. The finite state machine is usually represented as the transition table. The complexity of the algorithm for the average and the worst case performance is O(m+n).

2.3.1 Brute force

The brute force algorithm compares corresponding characters between patterns and text at all positions in any order. After each attempt, it shifts the pattern to the right by exactly one position. Then it starts the comparisons from the first character of pattern all over again. The brute force method requires no preprocessing and easy to understand and implement. But its worst-case running time of O (mn) is unsatisfactory. This can be improved because it makes a lot of comparisons that aren't actually necessary. In other words, the brute force method makes any possible comparisons even when there is no reason to believe a match will occur by shifting only one position to the right. As a result, this naive method is very slow in practice especially for large texts and patterns (Feng Cao 2004)

2.3.2 Knutt-Morris Pratt

Knuth-Morris-Pratt algorithm (Knuth M et al 2004) is the first linear running-time exact string matching algorithm. Although it is rarely the method of choice, it has historical importance and has been generalized to solve the problem of searching for a set of patterns in a text in time linear in the size of the text. By remembering some portions of the text that match the pattern, Morris and Pratt (Morris et al 1970) came up with an algorithm which runs in O(n + m). KMP does comparisons from left to right.

2.3.3 Boyer-Moore

In 1977, the biggest break-through in string matching was contributed by Boyer and Moore (Boyer and Moore 1977). Generally, the Boyer Moore method is much faster than the Knuth Morris Pratt method. As the size of the text string increases, the difference becomes much more apparent. So far Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. By introducing a pattern preprocessing stage (D. Gusfield 1999) the Boyer-Moore algorithm is able to analyze deeply the internal structure of a pattern and thus can shift the pattern more efficiently. The Boyer-Moore algorithm compares the pattern with the target text character-by-character from right to left. Once a mismatch occurs, it uses either the bad character rule or the good suffix rule, whichever contributes to a larger shift, to shift the pattern (Rytter, W. A 1982).

2.3.4 Karp Rapin

The Karp-Rabin algorithm (R. M. Karp & M. O. Rabin 1987) computes the hash function of each n-character substring in the text and checks if it equals the hash function of the pattern, where n is the length of the pattern. Karp and Rabin found an efficient way to compute a re-hash function for Text [i + 1 ... i + n] based on the hash value of Text [i ... i + n - 1], which makes the whole algorithm more efficient.

The advantage of the Karp-Rabin algorithm is that it can produce the same theoretical time bounds as the previously mentioned algorithms and requires a smaller number of registers. It is also conceptually very simple and therefore easy to program (R. M. Karp & M. O. Rabin 1987).

2.3.5 Performance Comparisons

Algorithm	**Preprocessing time**	**Matching time**
Naïve string search algorithm	0 (no preprocessing)	O(n m)
Rabin-Karp string search algorithm	θ(m)	Average O(n+m), Worst O(n m)
Knuth-Morris-Pratt algorithm	θ(m)	θ(n)
Boyer-Moore string search algorithm	O(m)	Average O(n/m), Worst O(n m)

Where m and n are the length of the two strings being compared.

Table 2.1 String searching algorithms

2.3.6 Applicability of string searching algorithms in Bio-sequence databases

From the above, there exist many algorithms that allow searching a string (pattern) inside another string as mentioned above. However, in bio-sequence databases exact similarity of sequences is rare. In addition to establishing a fact that one string is a substring of another, we need a measure of similarity between sequences. A standard example of such a situation is the case of searching similarities in biological sequences. In such a case algorithms like (M. Karp & M. O. Rabin 1987) will not produce needed results. Therefore, different algorithms solutions are required which do' approximate string matching in addition to exact string matching i.e. (shows a degree of similarity between strings/sequences). In bioinformatics sequence matching applications, exact matching is not always relevant. It is often more important to find sequences that match a given pattern in a reasonably approximate manner.

2.4 Sequence Alignment

Alignment based methods such as those based on Dynamic programming Needleman, smith (C. Setubal & J. Meidanis 1997), FASTA (William R. Pearson and David J. Lipman, 1988) and BLAST (Altschul et al 1990) have been developed for identifying sequence similarity. BLAST has been widely used by biologists for sequence analysis (Altschul et al 1990). These tools are largely dependent on heuristics, and they fail on queries which do not have very similar sequences deposited in the database. BLAST gives approximately accurate results and its speed is therefore achieved at the cost of some degree of precision (Clare Sansom 2000). Also alignment based methods suffer from the drawback of the increasing computational complexity with the increase in the number of the sequences as well as the size of the sequences. Therefore the research into alignment free sequence analysis is emerging at a greater speed than before. Also this is necessary to overcome critical limitations of sequence analysis by alignment. In (Susana Vinga & Jonas Almeida 2003) it has been mentioned that although the pace of work in this area is increasing sharply, the total number of published reports proposing or using alignment-free metrics is relatively small, still under the one hundred mark.

Effective algorithms for comparing protein and DNA sequences have been available for more than thirty years, since the publication of a global sequence comparison algorithm by (Needleman & Wunsch 1970). Global sequence comparison algorithms seek to align every residue in one sequence with every residue in a second, in contrast to the more commonly used local sequence alignment algorithms, which seek only the strongest region of similarity between two sequences. Global alignment algorithms are used for aligning families of sequences with similar lengths in preparation for phylogenetic analysis; global alignment scores can be transformed to the distance measures used for building evolutionary trees. Global similarity scores are rarely used to infer homology however, because the distribution of global similarity scores is not well understood and thus it is difficult to assign a statistical significance to a global similarity score. Moreover, many proteins are made up of domains that are homologous only over a portion of the protein sequence (William R. Pearson & Todd C. Wood 2000).

The notion of distance between sequences was first formalized by (Levenshtein 1965), who introduced 'edit distance'. This distance is defined for two strings as the smallest possible total cost of insertions, deletions, and replacements of characters that transform the first string to the second string. Some variants of the edit distance allow for reversals of sub-strings.

The edit distance problem is closely related to the problem of sequence alignment. The problems are essentially equivalent, and their results provide distinct views of the difference between two sequences. An edit transcript describes a process, whereas a sequence alignment provides a comparison. In particular, the alignment can be easily produced from a set of insertions and deletions of characters, and vice versa. In the context of biological sequences, similarity and alignment were first studied by (Needleman & Wunsch 1970). The Needleman-Wunsch sequence similarity and sequence alignment are global; i.e. this is an alignment of full length sequences. In practice, only extremely similar sequences can be nicely aligned globally. However, many proteins exhibit strong local similarity. The local alignment problem was studied by (Smith & Waterman 1981). Algorithms for calculating either local or global similarity for protein sequences do not assign equal cost to all possible steps. Rather, scoring matrices are used to give different weight to replacement of various pairs of characters. The most commonly used scoring matrices are BLOSUM (Henikoff 1992), and the older PAM (Dayhoff et al 1978).

Currently, the most popular algorithm and software package used for global and local similarity calculation is BLAST, along with a variant called PSI-BLAST. Other useful algorithms are FASTA and the Smith-Waterman dynamic programming algorithm (Ori Sasson 2005).

There are basically two types of sequence alignment algorithms, local and global. Both local and global alignment algorithms are based on the dynamic programming approach. Smith-Waterman (SW) local alignment matches two sequences for a fixed length for which they show a high degree of similarity. However, the Needleman-Wunsch (NW) algorithm is a global alignment algorithm which compares two sequences over their entire lengths. It seeks to match as many nucleotides or amino acids as possible between the sequences to give the optimal alignment. It is considered suitable for finding the best alignment for two sequences which are of similar length (Dzivi PS 2008).

The Smith-Waterman (SW) algorithm is a time demanding algorithm because it searches for optimal local alignments; it also imposes some requirements on the computer's memory resources as the comparison takes place on a character-to-character basis.

Today's research requires fast and effective data analysis. BLAST and FASTA are heuristic approximations of the SW algorithms and are less time consuming; therefore they are preferred over SW algorithm. However, these approximations are less sensitive and less optimal. SW algorithm has been implemented on different hardware architecture and with different algorithmic approaches (Dzivi PS 2008).

Alignment algorithms are used to find similarity between biological sequences, such as DNA and proteins. By aligning a sequence with a database, similar sequences can be found. These can be used to identify the source of a query sequence, to find commonalities between organisms, or to infer an ancestral relation. Various methods of performing biological sequence alignment exist, including dynamic programming and heuristic methods. Dynamic programming methods are guaranteed to find all optimal alignments, but are relatively slow; heuristic methods are faster but less precise (M.A. Kentie 2005).

The algorithms we will discuss fall into two categories.

1. Dynamic programming algorithms that find exact optimal solutions for a given scoring system.
2. Heuristic algorithms motivated by the specific problem of similarity search against large sequence databases.

Dynamic Programming Algorithms	
1970	Needleman Wunsch
1981	Smith Waterman
Heuristic Algorithms	
1990	BLAST
1988	FASTA

Table 2.2 Dynaming programming and Heuristic algorithms

2.4.1 Needleman-Wunsch Algorithm

The Needleman-Wunch (NW) algorithm (Needleman & Wunsch 1970) is a nonlinear global optimization method that was developed for amino acid sequence alignment in proteins. This was the first of many important alignment techniques which now find application in the Human Genome Project.

Needleman-Wunsch uses dynamic programming in order to obtain global alignment between two sequences. Global alignment, as the name suggests takes into account all the elements of the two sequences while aligning the two sequences. We can also call it as an "end to end" alignment. In Needleman-Wunsch algorithm, a scoring matrix of size m*n (m being the length of longer sequence and n being that of the shorter sequence) is first formed. The optimal score at each matrix position is calculated by adding the current match score to previously scored positions and subtracting gap penalties. Each matrix position may have a positive, negative or 0 value. For two sequences

$S_1 = a_1 a_2 \ldots\ldots\ldots\ldots\ldots\ldots a_m$

$S_2 = b_1 b_2 \ldots\ldots\ldots\ldots\ldots\ldots b_n$

Where $T_{ij} = T(a_1 a_2 \ldots\ldots\ldots\ldots\ldots\ldots a_m, b_1 b_2 \ldots\ldots\ldots\ldots\ldots\ldots b_n)$ then the element at the i,j^th position of the matrix T_{ij} is given by

$$T_{ij} = \text{Max}\left\{ \begin{array}{l} T_{i-1,j-1}+s, \\ \text{Max}(T_{i-x,j} - p_x),\ x >= 1 \\ \text{Max}\ (T_{i,j-y} - p_y), y >= 1 \end{array} \right\}$$

Where T_{ij} is the score at position i in the sequence S_1 and j in the sequence S_2, $T(a_i\ b_j)$ is the score for aligning the characters at positions i and j, p_x is the penalty for a gap of length x in the sequence S_1 and p_y is the penalty for a gap of length y in the sequence S_2. After the T matrix is filled up, to determine an optimal alignment of the sequences from scoring matrix, a method called trace back is used. The trace back keeps track of the position in the scoring matrix that contributed to the highest overall score found. The positions may align or may be next to a gap, depending on the information in the trace back matrix. There may exist multiple maximal alignments (V. Anitha & B.Poorna 2005).

2.4.2 Smith-Waterman algorithm

The Smith-Waterman algorithm is a dynamic programming method for determining similarity between nucleotide or protein sequences. The algorithm was first proposed in

1981 by Smith and Waterman and is identifying homologous regions between sequences by searching for optimal local alignments. To find the optimal local alignment, a scoring system including a set of specified gap penalties is used (Smith & Waterman 1981). Homology identified by sequence database searches often implies shared functionality between sequences and further research and development might depend on the accuracy of the search results. The Smith-Waterman algorithm is build on the idea of comparing segments of all possible lengths between two sequences to identify the best local alignment. This means that the Smith-Waterman search is very sensitive and ensures an optimal alignment of the sequences. Unfortunately, this also has the effect that the method is both time and CPU intensive.

The Smith-Waterman algorithm is a method of database similarity searching which considers the best local alignment between a query sequence and sequences in the database being searched. The Smith-Waterman algorithm allows consideration of indels (insertions/deletions) and compares fragments of arbitrary lengths between two sequences and this way the optimal local alignments are identified (Smith & Waterman 1981).

It has three main steps

1. Assign similarity values
2. For each cell, allowing insertions and deletions give the maximum possible scoring value
3. Construct an alignment (pathway) back from the highest scoring cell

The Smith-Waterman Algorithm is a variation of the Needleman-Wunsch Algorithm that applies more readily to the alignment of strings with different lengths. The two additions that the Smith-Waterman Algorithm includes are (Polanski, Andrzej & Kimmel Marek 2007):

1. If an optimal cumulative score becomes negative then it is reset to zero.
2. The starting point of the alignment occurs at the largest score in the optimal cumulative

These additions do not change the time complexity if we consider matrices of equal length but the Smith-Waterman algorithm is quite useful in finding unique local alignments of strings (Polanskiet al 2007). The draw backs for the Smith-Waterman Algorithm are however the same as that of the Needleman-Wunsch Algorithm and are not useful when implemented on large databases which leads us to the Heuristic approaches (Jennifer Johnstone 2008).

2.4.3 BLAST

BLAST is an acronym for basic local alignment search tool; the BLAST family of database search programs takes as input a query DNA or protein sequence, and search DNA or protein sequence databases for similarities that may indicate homology. BLAST (Basic Local Alignment Search Tool) employs a heuristic algorithm to search a sequence database for the closest matches to a test sequence (Altschul et al 1990).

BLAST (Basic Local Alignment Search Tool) also identifies homologous sequences by database searching (Altschul et al 1990). BLAST identifies the local alignments between sequences by finding short matches and from these initial matches (local) alignments are created. The BLAST algorithm is a development of the Smith-Waterman algorithm suggesting a time-optimized model contrary to the more accurate but time consuming calculations of the Smith-Waterman algorithm (Altschul et al 1990).

2.4.4 FASTA

Pronounced "FAST-A," The **FASTA** algorithm, developed by (Pearson & Lipman 1988), proceeds along almost exactly these lines. It breaks up the query and database into smaller words, observes the pattern of word-to-word matches of a given length, and marks potential matches before performing a more sensitive search using local sequence alignment.

First, FASTA first breaks the query and database sequence into k-mer words, then hashes the locations of the database words to speed up later searches. For each k-mer in the query, FASTA finds an exact match in the hashed database and extends it in both directions, allowing for some mismatches. It then scores these matches using BLOSUM or PAM and uses dynamic programming to co-linearly stitch the high-scoring matches together.

2.5 Challenges of sequence alignment algorithms

As the sizes of bio-sequence databases increase, searching a query sequence against a database becomes increasingly time consuming. Significant gains in speed can be achieved by employing heuristic search methods such as BLAST (Altschul *et al* 1990), rather than slower optimal alignment methods such as the Smith–Waterman algorithm (Smith & Waterman 1981). However, sensitivity is thereby lost (Pearson 1995; Shpaer *et al* 1996), and although the fraction of hits missed by employing a heuristic method may be small (Brenner *et al* 1998), the distant homologs thereby missed may be among the hits of greatest interest. In addition, on hardware specialized for dynamic programming searches, a gain in speed from adopting heuristic methods may not be possible. Although such hardware may be quite fast, certain types of search are still time consuming and further acceleration of the search may be desirable.

Extensive work and research has been in sequence similarity search and has mainly focussed on sequence alignment based approach. Sequence alignment method, though very powerful and popular, leave space for further research, to optimize the computational complexity arising out of the alignment process (Maulika S Patel & Himanshu S Mazumdar 2010).

Alignment based methods such as those based on Dynamic programming Needleman, smith] (C. Setubal & J. Meidanis, 1997), FASTA (William R. Pearson & David J. Lipman 1988) and BLAST (Altschul S. F. et al 1990) have been developed for identifying sequence similarity. BLAST has been widely used by biologists for sequence analysis (Altschul S. F. et al 1990). These tools are largely dependent on heuristics, and they fail on queries which do not have very similar sequences deposited in the database. BLAST gives approximately accurate results and its speed is therefore achieved at the cost of some degree of precision (Clare Sansom 2000). Also alignment based methods suffer from the drawback of the increasing computational complexity with the increase in the number of the sequences as well as the size of the sequences. Therefore the research into alignment free sequence analysis is emerging at a greater speed than before. Also this is necessary to overcome critical limitations of sequence analysis by alignment. In (Susana Vinga & Jonas Almeida 2003) it has been mentioned that although the pace of work in this area is increasing sharply, the total number of published reports proposing or using alignment-free metrics is relatively small, still under the one hundred mark.

2.5 Edit distance Algorithms

The edit distance between two strings over a fixed alphabet ∑ is the minimum cost of transforming one string into the other via a sequence of character deletion, insertion, and replacement operations (R.Wagner and M.Fischer 1974). The cost of these elementary editing operations is given by some scoring function which induces a metric on strings over ∑. The simplest and most common scoring function is the Levenshtein distance (V.I. Levenshtein 1966),which assigns a uniform score of 1 for every operation. Determining the edit-distance between a pair of strings is a fundamental problem in computer science in general, and in combinatorial pattern matching in particular, with applications ranging from database indexing and word processing, to bioinformatics (D. Gusfield 1997)

The edit distance like algorithms are used to compute a distance between DNA sequences (strings overA, C, G, T, or protein sequences (over an alphabet of 20 amino acids), for various purposes, e.g.:

- To find genes or proteins that may have shared functions or properties
- To infer family relationships and evolutionary trees over different organisms

Searching sequences while allowing for a certain number of insertions, deletions, and substitutions, is however known to be a computationally expensive task, and consequently exact methods can usually not be applied in practice.

2.5.1 Edit Distance and Bio-sequence similarity

Edit distance derives its definition from the concept of mutations by assigning weight to each mutation. Given two sequences the distance between them is the minimal sum of weights for the set of mutations that transform one sequence to another. For similarity we assign weights corresponding to base resemblance. Given two sequences the similarity between them is the maximal sum of such weights.

2.5.2 Levenhstein distance Algorithm

Levenshtein distance is named after the Russian scientist Vladimir Levenshtein, who devised the algorithm in 1965. The Levenshtein distance between two strings is given by the minimum number of operations needed to transform one string into the other, where an operation is an insertion, deletion, or substitution of a single character. Levenshtein distance (LD) is a measure of the similarity between two inputs: the source *s* and the target input *t*. The distance is the number of deletions, insertions, or substitutions required to transform *s* into *t*. For example, If s is "math" and t is "math", then LD(s,t) = 0, because no transformations are needed. If s is "math" and t is "mats", then LD(s,t) = 1, because one substitution (change "h" to "s") is sufficient to transform s into t. The more different the inputs are, the greater the Levenshtein distance is.Insertion, deletion and substitution are the main criteria for determining Levenshtein Distance. The position of a character plays an important role to determine the distance (Dr. Mashud Kabir 2009).

2.5.3 Damerau Levenhstein Distance Algorithm

The Damerau-Levenshtein distance between two strings is the minimum number of operations of the form:

- Substituting a character,
- Deleting a character,
- Inserting a character, or
- Swapping two characters in the original strings,

which transforms one string into the other. The function, by definition, is symmetric and distance(s, t) ≤ max(s.length(), t.length()) the longer string `t` can be transformed into the shorter string `s` by making s.length() switches and `t.length() - s.length()` deletions.

The Damerau-Levenshtein distance is a generalization of the Levenshtein distance by adding one additional operation: swapping two characters. In the Levenshtein distance, swapping two characters requires two operations, usually a deletion and an insertion. The code presented restricts swaps to two characters which are adjacent in the original strings. This restricted edit distance allegedly does not satisfy the triangle inequality (Damerau F.J 1964).

Damerau-Levenshtein distance comes from Levenshtein distance that counts transposition as a single edit operation. The Damerau-Levenshtein distance is equal to the minimal number of insertions, deletions, substitutions and transpositions needed to transform one string into the other. (Karen Kukich 1992) described several edit distance algorithms which use Damerau-Levenshtein distance. It has been proven that the use of Damerau-Levenshtein metric to calculate the similarity between two words is a slow process (Karen Kukich 1992).

In this study, we are proposing Damerau-Levenshtein distance algorithm to be used to perform sequence search.

CHAPTER 3: METHODOLOGY

3.1 Methodology

The research methodology is divided into six phases as shown below:

1. Study selected sequence comparison algorithms(sequence matching and sequence alignment algorithms)
2. Review these and analyze these algorithms
3. Propose one of the algorithms (Damerau Levenhstein algorithm) which use edit distance metric to make string comparison be applied for searching biological sequence databases.
4. Implement Damerau Levenhstein algorithm and enhanced it by implementing a parallel version of Damerau Levenshtein.
5. Extend these both algorithms by designing bio-sequence search engines using the concepts of World Wide Web search methods and techniques.
6. Apply the two bio-sequence search engines prototypes(i.e. one based on ordinary Damerau Levenshtein algorithm and the parallelized Damerau Levenshtein algorithm to demonstrate how it can be used to search biological sequence databases
7. Perform comparison of the two versions of the algorithm and measure the success of the bio-sequence search engine developed compared to currently existing tools mainly tools based on sequence alignment algorithms.

The two algorithms are implemented using Visual Studio- C#. The execution times of the serial and parallelized algorithms are taken from the outputs and then analyzed and compared against each other. We will test the performance of the two versions of the algorithm by simulating it over a wide range of bio-sequence sample size ie sequences with different lengths.The data set used is kept uniform throughout the experiments with the two algorithms. The search engine will take as inputs the sequence and the algorithm sensitivity (a percentage of accuracy). In the simulation the adjustable parameters will be the length of sequences and the percentage of sensitivity. The outputs of the algorithm are a list of sequences with some percentage of similarity and time it took to search the sequence in the database. The output will be the execution time is the performance measure.

3.2 Limitations of methodology

1. The data being used is hypothetical bio-sequence data
2. The study makes a comparison of the algorithm on one and two core machines only. Its ideal performance should be done using a quad processor and above to be able to ascertain performance improvement.
3. Complete search engine is not tested with huge practical bio-sequence databases.
4. Multi-threading programming overheads - Overhead costs associated with setting up and managing parallel programming features. If you have only a small amount of work to perform, the overhead can outweigh the performance benefit.
5. Coordinating Data - If your pieces of work share common data or need to work in a concerted manner, you will need to provide coordination. As a general rule, the more coordination leads, the poorer the performance of your parallel program.

3.3 Tools and Technologies

The tools and technologies used to build the web-interface are ASP.NET 4.0, VB.NET, IIS 6.0 Server, XML Web Services, Microsoft SQL Server 2005 and Microsoft Visual Studio 2010 IDE.

3.3.1 ASP.NET 4.0

ASP.NET 4.0 is the web-application framework developed by Microsoft that programmers can use to build dynamic websites, web-applications and XML web services. It is a part of Microsoft .NET. ASP.NET is built on Common Language Runtime (CLR), meaning programmers can write ASP.NET code in any Microsoft .NET language. The services provided by ASP.NET for building enterprise-class web-applications are: page and controls framework, the ASP.NET compiler, security infrastructure, state-management facilities, application configuration, debugging support, XML web services framework, extensible hosting environment and application life cycle management and extensible designer environment.

3.3.2 Internet Information Services

Internet Information Services is a web server included in Windows Operating System which provides World Wide Web publishing services, File Transfer Protocol (FTP) services, Simple Mail Transfer Protocol (SMTP) services and Network News Transfer Protocol (NNTP) services. It provides highly reliable and manageable infrastructure for web-applications. It is a high performance, secure and extensible web server provided by Microsoft. IIS 5.0 is built on features and capabilities needed to deliver web-applications required in an increasingly Internet centric business environment. It is easy to install, maintain and has features that make it reliable and better performing.

3.3.3 Microsoft SQL Server 2005

Microsoft SQL Server 2005 is a relational database management system and analysis system for e-commerce, line-of-business and data warehousing solutions. SQL Server 2005 is Microsoft's next generation data management and analysis software that will deliver increased scalability, availability, and security to enterprise data and analytical applications while making them easier to create deploy and manage. Its primary query language is Transact-SQL, an implementation of the ANSI/ISO standard SQL.

3.3.4 Microsoft Visual Studio 2010

Microsoft Visual Studio 2010 is a Microsoft's flagship software development product for computer programmers. It centers on an Integrated Development Environment which lets programmers create standalone applications, web sites, web-applications, and web services that run on any platforms supported by Microsoft's .NET framework. Visual Studio includes Visual Basic .NET, Visual C++, Visual C#, Visual J# and ASP.NET.

CHAPTER 4: DESIGN AND IMPLEMENTATION

4.1 Background Design

The design of Bio-sequence search engine discussed in this chapter is essentially based on the fact that the current sequence comparison algorithms architectures do not scale cost-effectively because they rely on sequence alignment algorithms to perform sequence comparisons between a query sequence and a database of sequences. The current methods utilizes dynamic programming and heuristic methods (Discussed in chapter 3).

The Damerau-Levenshtein distance is defined as the minimum number of primitive edit operations needed to transform one string into the other. The edit operations are:

- Substitution
- Deletion
- Insertion
- Transposition of two adjacent characters.

Edit distance primitive operations	Equivalent Biological mutations
Insertion	Insertion of a nucleotide
Deletion	Deletion of a nucleotide
Substitution	Substitution of a nucleotide
Transposition of two adjacent characters.	Transposition of two adjacent nucleotides

Table 4.1 Edit distance operations

These operations are equivalent to bio-sequence mutations which are common in DNA and proteins. DNA frequently undergoes mutations (insertions, deletions, substitutions, and transpositions); therefore Damerau–Levenshtein distance is an appropriate metric of comparison between bio-sequences. Damerau Levenhstein distance algorithm best models all types of biological mutations which are common biological phenomenon. This project aims to model this algorithm by creating a search tool for bio-sequences.

To better enhance its performance, we have implemented a parallelized version of Damerau Levenhstein distance algorithm using multi-threading programming model. A comparison of ordinary Damerau Levenhstein distance algorithm and parallelized version is then performed to ascertain performance improvement.

This is a different new approach as much of the research worked done before have focused on sequence alignment algorithms which can be group into two approaches:

3. Dynamic programming such as Needleman-Wunsch or Smith-Waterman
4. Heuristic methods such as BLAST and FASTA

Dynamic programming methods are guaranteed to find all optimal alignments, but are relatively slow; heuristic methods are faster but less precise (M.A. Kentie 2005).

4.2 Parallelism through multithreading

Multithreaded programs are similar to the single-threaded programs that are associated with single path of execution. They differ only in the fact that they support more than one concurrent thread of execution-that is, they are able to simultaneously execute multiple sequences of instructions. Each instruction sequence has its own unique flow of control that is independent of all others. These independently executed instruction sequences are known as threads (Kevin Haghighat 2008).

4.3 Benefits of multithreading

Traditionally, programs are single-path execution, hence a single thread. This practice would have made the production of today's software production impossible as the need of speed required programs to perform multiple tasks and events at the same time. With traditional turn-by-turn game, such as tic-tac-toe or chess, the traditional approach works fine, however with new age multitasking programs where multiple events need to run in parallel, the traditional approach proves useless (Kevin Haghighat 2008).

The benefits of multithreading can be summarized as follows:

Responsiveness: Multithreading allows a process to keep running even if some threads within the process are stalled, working on a lengthy task, or awaiting user interaction. Using a digital alarm clock as an example of a process, the thread of keeping track of time, continues while an alarm is sounding while another awaits it's time to activate.

Cost Effective: Memory and resource allocation to process creation remains costly where as threads share the resources allocated to the process they reside in making it less costly to make threads or move them from one process to another.

Resource Distribution: The inherit property of sharing memory and resources of the parent process fosters the ability of having multiple treads occupying the same address space.

Cross-Processor Distribution: The benefits of multithreading are multiplied as the number of available processors increase opposite to single threading where only one processor is used. In a multiprocessor architecture, running of threads can distribute across multiple processors in parallel thereby increasing efficiency (Kevin Haghighat 2008).

4.4 System Architecture

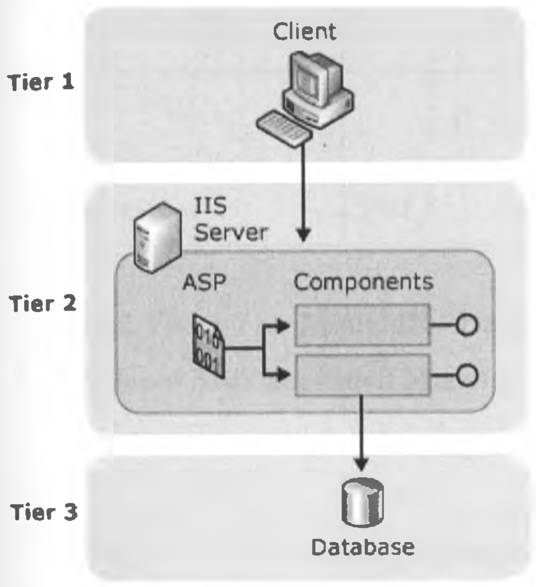


Figure 4.1Three Tier Model

Task	Tier	Description
User interface and navigation	Tier 1	This layer provide a graphical user interface (GUI) so that users can interact with the application, input data, and view the results of requests, it also manages the manipulation and formatting of data once the client receives it back from the server. A web browser performs the tasks of this layer.
Business logic	Tier 2	Business logic, which involves the rules that govern application processing, connects the user in tier 1 with the data in tier 3. The functions that the rules govern closely mimic everyday business tasks, and can be a single task or a series of tasks
Data services	Tier 3	Data services are provided by a structured (SQL database, XML database) which manages and provides access to the data contained within.

Table 4.2 Three Tier Model Description

4.5 Proposed Multithreaded Model for Damerau Levenhstein distance algorithm

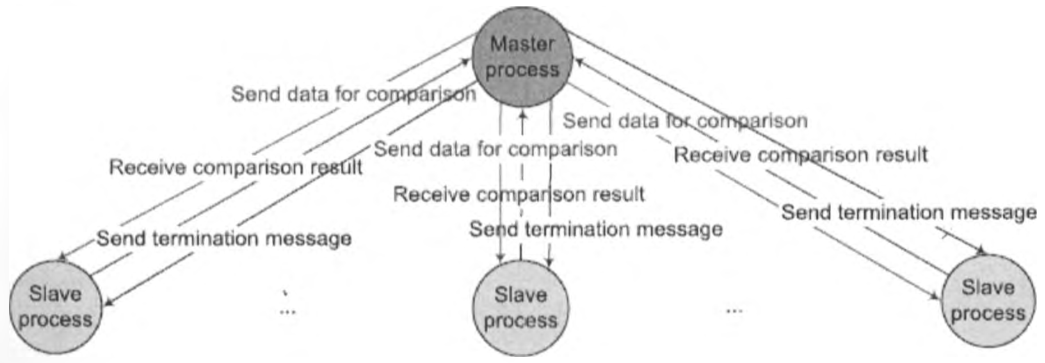


Figure 4.2 Proposed Multithreaded Model for Damerau Levenshtein algorithm

The master process is responsible for the following tasks:

1. Sends source sequence S to the slave processes;
2. Partitions the target sequence T;
3. Sends a part of sequence T to each slave process;
4. Receives comparison results from the slave processes;
5. Sends termination message when M sequence is fully processed.

Each slave processes performs the following operations:

1. Receives part of target sequence T from the master process;
2. Processes sequence T part comparing it with the source sequence S;
3. Sends the comparison result to the master process;
4. Terminates when a termination message is received.

The processing stage of the computational model requires each slave process independently to compare the source sequence and part of the target sequence following Damerau Levenshtein distance algorithm. The communication stage requires the master process to distribute data to the slave processes, gather the results and send new parts of the target sequence to the slave processes. Communication between the master and the slave processes is performed using synchronous communication.

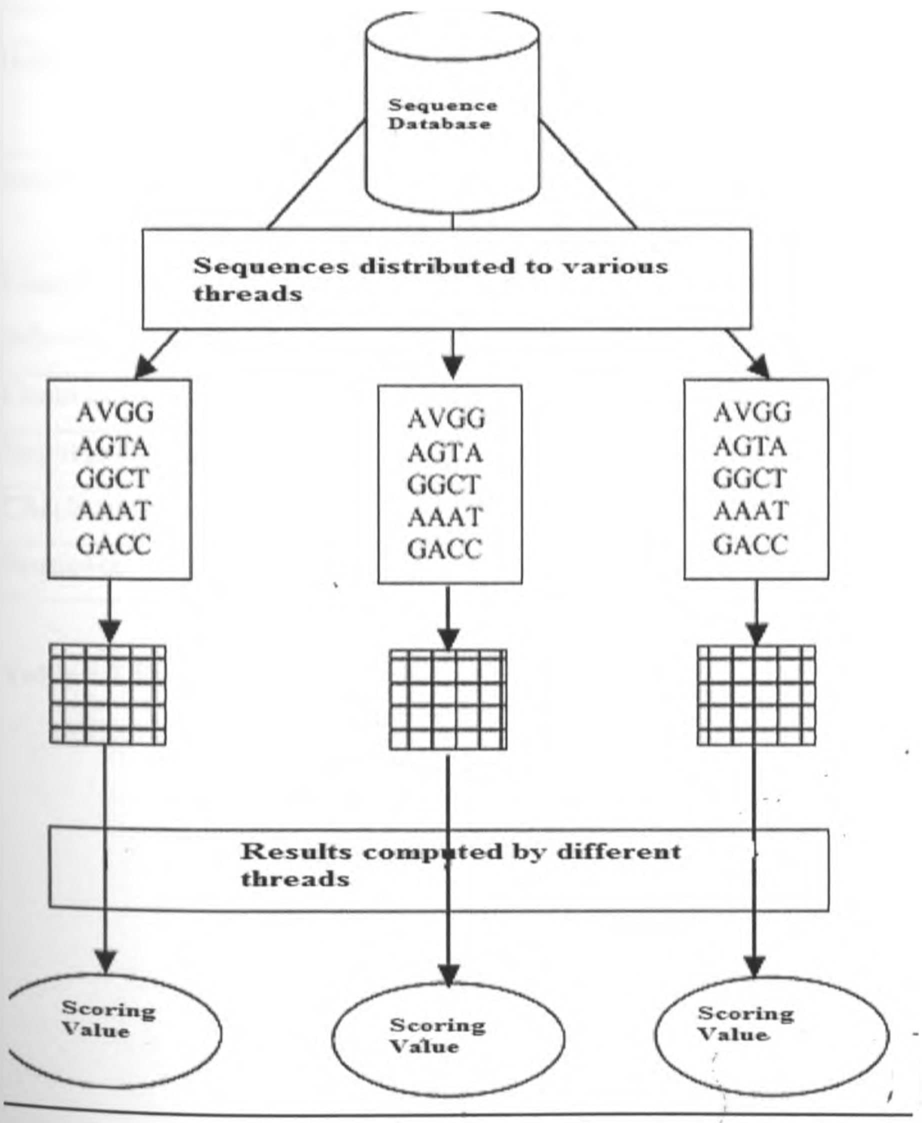


Figure 4.3 Detailed parallel model

4.6 Back End (Database)

Bio-sequence details are stored in the SQL Server database. These details are stored as tables in the database where tables are related with other tables using the primary key-foreign key relationship. The table details are shown in Table 7.1.

Sequence Table

FIELD	DESCRIPTION	DATA TYPE
Name	ID is a unique identifier	VARCHAR
Definition	A brief one line textual sequence description	VARCHAR
Accession Number	A Constant data identifier.	VARCHAR
Complete literature references	Literature references	VARCHAR
Comments & keywords	Comments and keywords	VARCHAR
Important Feature	All important features	VARCHAR
Checksum	Checksum line	VARCHAR
Sequence	Sequence line	VARCHAR

Table 4.3 Sequence table structure

4.7 Pseudo code for Damerau Leveinshtein edit distance algorithm

```
int DamerauLevenshteinDistance(char str1[1..lenStr1], char str2[1..lenStr2])
  // d is a table with lenStr1+1 rows and lenStr2+1 columns
  declare int d[0..lenStr1, 0..lenStr2]
  // i and j are used to iterate over str1 and str2
  declare int i, j, cost

  for i from 0 to lenStr1
     d[i, 0] := i
  for j from 1 to lenStr2
     d[0, j] := j

  for i from 1 to lenStr1
     for j from 1 to lenStr2
        if str1[i] = str2[j] then cost := 0
                          else cost := 1
        d[i, j] := minimum(
                          d[i-1, j  ] + 1,     // deletion
                          d[i  , j-1] + 1,     // insertion
                          d[i-1, j-1] + cost   // substitution
                       )
        if(i > 1 and j > 1 and str1[i] = str2[j-1] and str1[i-1] = str2[j]) then
           d[i, j] := minimum(
                          d[i, j],
                          d[i-2, j-2] + cost   // transposition
                       )

   return d[lenStr1, lenStr2]
```

Parallize version pseudo code

```
For each slave i
        Master ⟶  send an appropriate part of string s to slave i
        Slaves ⟶    receive part of (s) and process it
While not end of(s)
Slave j    ⟶   sends an outcome of processing the part of (s) to master
Master     ⟶   receive an outcome from slave j
Master     ⟶   sends a part of string s to slave j
Slave j    ⟶   receives part of (s) and processes it
End while
```

4.8 Damerau Leveinshtein edit distance algorithm analysis

This algorithm calculates the cost of the so-called optimal string alignment, which does not always equal the edit distance. It is also easy to see that the cost of the optimal string alignment is the number of edit operations needed to make the strings equal under the condition that no substring is edited more than once. We will also call this value a restricted edit distance. As noted by (G. Navarro 2001) in the general case, i.e. when a set of elementary edition operations includes substitutions of arbitrary length strings, unrestricted edit distance is hardly computable. However, the goal is achievable in the simpler case of Damerau-Levenshtein distance. It is also possible to compute unrestricted distance treating reversals of arbitrary, not obligatory adjacent characters as a single edit operation. To devise a proper algorithm to calculate unrestricted Damerau-Levenshtein distance algorithm, there always exists an optimal sequence of edit operations, where once-transposed letters are never modified afterwards. Thus, we need to consider only two symmetric ways of modifying a substring more than once: (1) transpose letters and insert an arbitrary number of characters between them, or (2) delete a sequence of characters and transpose letters that become adjacent after deletion. The straightforward implementation of this idea gives an algorithm of cubic complexity: O (M.N.max (M, N) where M and N are string lengths. Using the ideas of (Lowrance & Wagner 1975) this naive algorithm can be improved to be O (M.N) in the worst case [http://cosmopedia.net/Damerau-Levenshtein_distance].

4.9 Performance Calculation

```
int LevenshteinDistance(char s[1..m], char t[1..n])
 { // mat is a matrix with m+1 rows and n+1 columns
declare int mat[0..m, 0..n]
for i from 0 to m --------------------------------------------------------------0(m)
                mat[i, 0] := i // deletion
for j from 0 to n --------------------------------------------------------------0(n)
              mat[0, j] := j // insertion
for j from 1 to n { ------------------------------------------------------------0(n)
       for i from 1 to m { -------------------------------------------------0(m)
                if s[i] = t[j] then
                mat[i, j] := mat[i-1, j-1]
                else
                    mat[i, j] := minimum ( mat[i-1, j] + 1, // deletion
                    mat[i, j-1] + 1, // insertion
                    mat[i-1, j-1] + 1 // substitution ) } }
return mat[m, n] }
```

Performance Calculation

Therefore,

$$T(n) = n + m + (n * m)$$
$$= n + m + nm$$

Since,

nm is the highest value.

Therefore,

$T(n) = 0(nm)$

Where n and m are the lengths of the strings.

5.0 Implementation

The objective of this project is to develop an online bio-sequence search engine using a parallelized version of Damerau Levenshtein distance algorithm. When the user types a biological sequence in the user interface, a Web Server is contacted to get the requested information. In the .NET Framework, IIS (Internet Information Service) acts as the Web Server. The sole task of a Web Server is to accept incoming HTTP requests and to return the requested resource in an HTTP response. The first thing IIS does when a request comes in is to decide how to handle the request. Its decision is based upon the requested file's extension. For example, if the requested file has the .asp extension, IIS will route the request to be handled by asp.dll. If it has the extension of .aspx, .ascx, etc, it will route the request to be handled by ASP.NET Engine. The ASP.NET Engine then gets the requested file, and if necessary contacts the database through ADO.NET for the required file and then the information is sent back to the Client's browser. Figure 21 shows how a client browser interacts with the Web server and how the Web server handles the request from client (Swapna Kodali 2007).

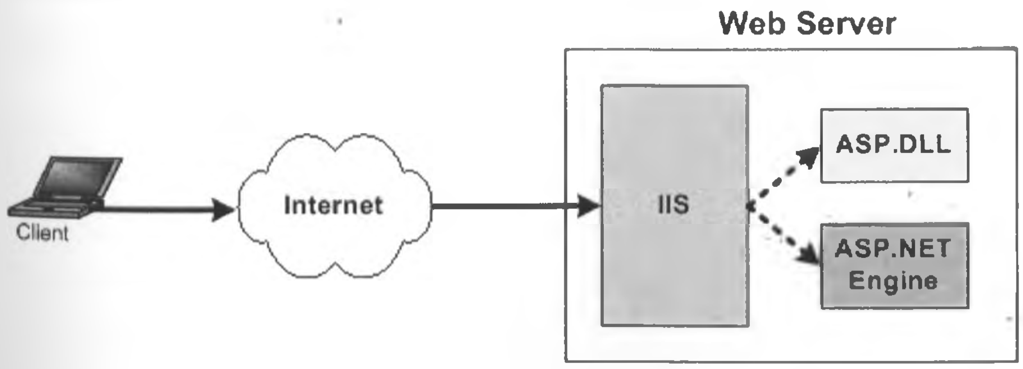


Figure 4.3 Client browser Web server interactions

5.1 Platform

In this research, local machine (HP Intel(R) Core™ 2 Duo Processors, CPU (1.80 GHz), 2 GB RAM and Windows XP Operating System) were used with Microsoft Visual Studio.Net software 2010 version.The web-interface is developed using Microsoft ASP.NET and XML Web Services with Internet Information Services (IIS) as the web server. The report is written in detail with description of implementation, tools used and details of testing, results obtained and conclusions drawn. A brief description of the future work to be conducted in this area is also illustrated.

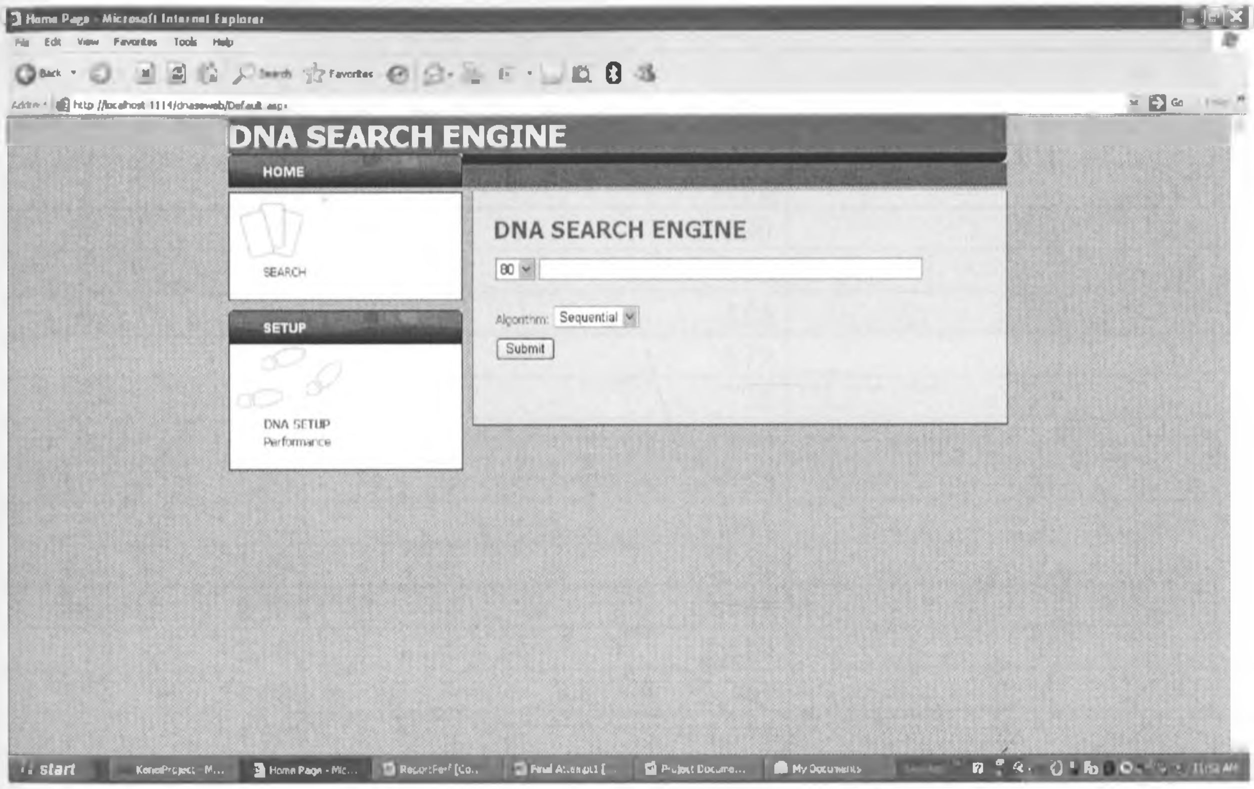


Figure 4.4 Search Engine Graphical User Interface

CHAPTER 5: EXPERIMENTAL RESULTS AND DISCUSSION

5.1 Presentation of results

We now present the results of running the parallel multithreaded and the serial Damerau Levenhstein distance algorithm implementations. Tests are carried out on single processor machine (one core system) and two processor machine (two core system). In both cases the parallelized version performances is faster than the serial counterpart.

5.1.1 Results - Single Processor Machine

No	Sample Size	Serial Damerau Levenshtein distance algorithm	Parallel Damerau Levenshtein distance algorithm
1	100	101	66
2	200	197	112
3	300	292	167
4	400	390	216
5	500	486	262
6	600	584	310
7	700	679	367
8	800	775	407
9	900	870	525
10	1,000	957	556
11	1,500	1,444	839
12	2,000	1,926	1,129
13	2,500	2,407	1,281
14	3,000	2,889	1,637
15	3,500	3,363	1,992
16	4,000	3,848	2,076
17	4,500	4,321	2,588
18	5,000	4,809	2,898
19	5,500	5,305	2,906
20	6,000	5,699	3,465
21	6,500	6,185	3,583
22	7,000	6,663	4,171
23	7,500	7,475	4,315
24	8,000	7,944	4,753
25	8,500	8,216	4,899
26	9,000	8,785	5,830
27	9,500	9,204	5,500
28	10,000	9,616	6,080

Table 5.1.1 Results performance on uni-processor machine

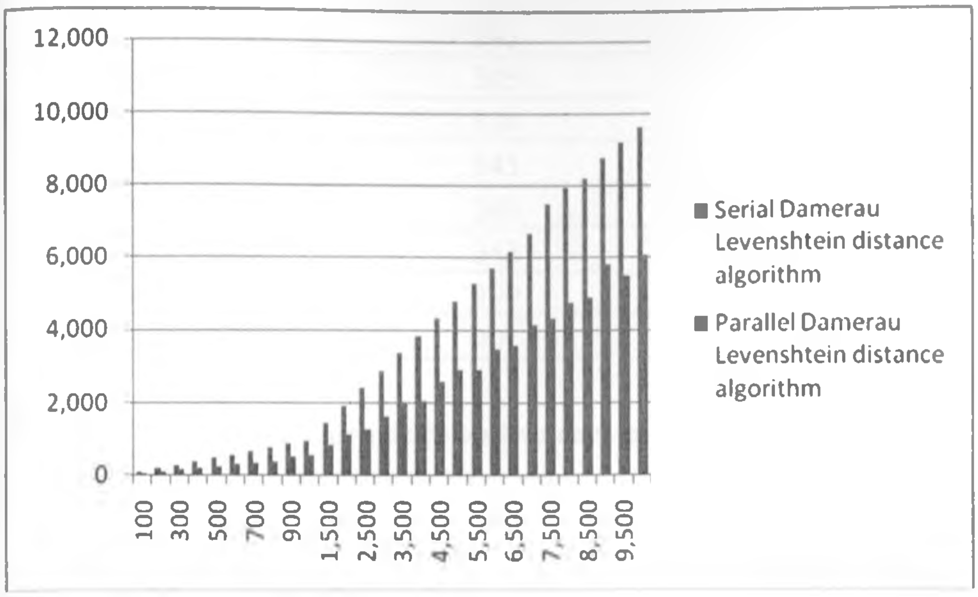


Figure 5.1.1 Performance Comparison

5.1.2 Results - Two core processor Machine

No	Sample Size	Serial Damerau Levenshtein distance algorithm	Parallel Damerau Levenshtein distance algorithm
1	100	14	9
2	200	16	12
3	300	21	19
4	400	27	18
5	500	31	21
6	600	34	24
7	700	39	26
8	800	42	31
9	900	46	36
10	1,000	67	42
11	1,500	89	69
12	2,000	107	71
13	2,500	125	132
14	3,000	145	152
15	3,500	164	161
16	4,000	184	172
17	4,500	203	176
18	5,000	222	187
19	5,500	243	199
20	6,000	266	212

21	6,500	285	234
22	7,000	305	269
23	7,500	324	289
24	8,000	345	293
25	8,500	360	302
26	9,000	383	321
27	9,500	399	339
28	10,000	401	364

Table 5.1.2 Results performance on two core processor machine.

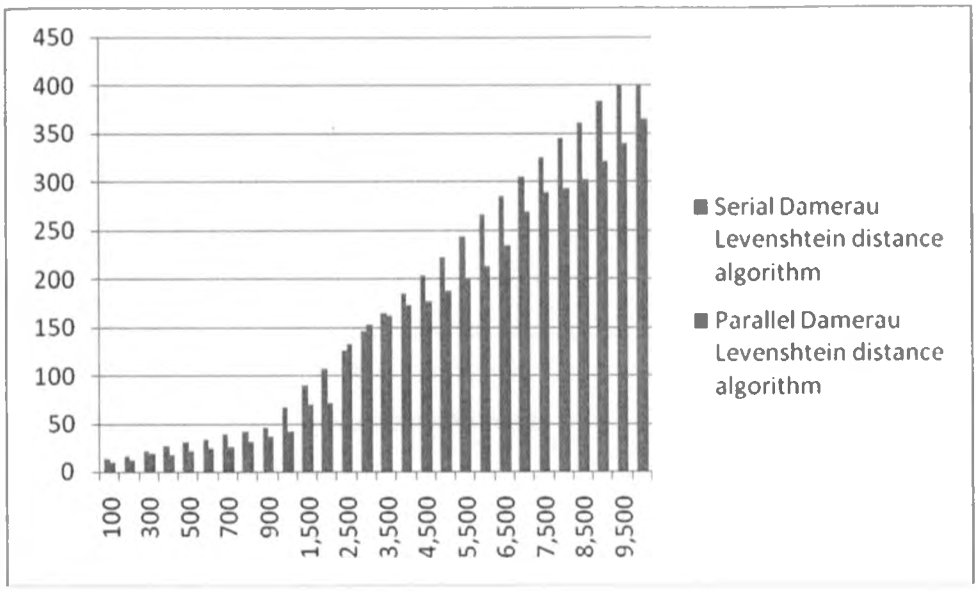


Figure 5.1.2 Performance comparison

5.4 Performance under different number of threads

5.4.1 One processor machine

No. of threads	t100(s)	t200(s)	t500(s)
1	0.12	0.995	40.14
2	0.062	0.5	20.27
4	0.062	0.501	20.34
8	0.064	0.504	20.47

Table 5.4.1 One core processor performance of threads

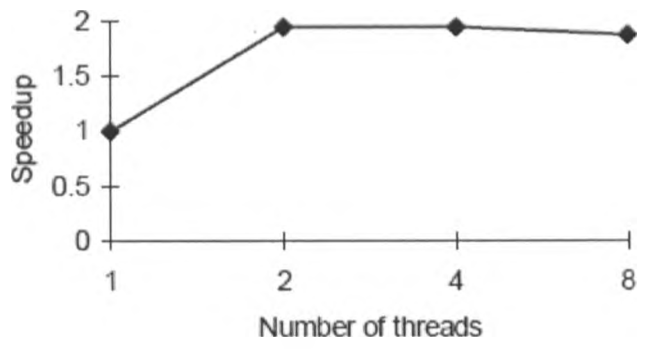


Figure 5.4.1 One core processor performance of threads

5.4.2 Two core processor machine

No. of threads	t100(s)	t200(s)	t500(s)
1	0.053	0.387	6.504
2	0.058	0.333	4.181
4	0.082	0.354	4.321
8	0.162	0.402	4.642

Table 5.4.2 Two processor machine performance of threads

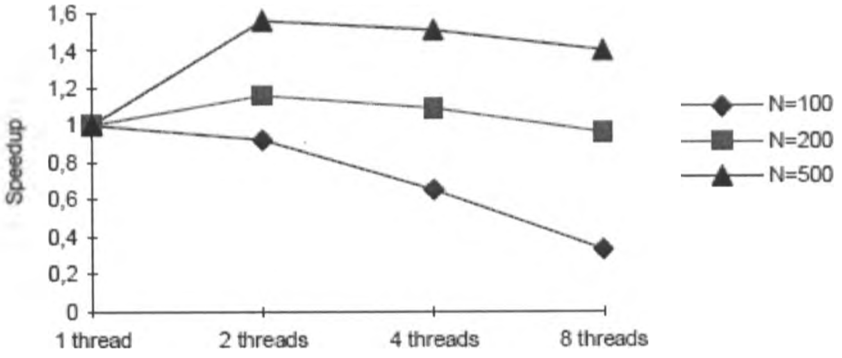


Figure 5.4.2 Two processor machine performance of threads

5.2 Discussion of results

5.2.1 Experimental results of serial and Multi-threaded algorithm

The multithreaded implementation of Damerau Levenhstein distance algorithm (both single threaded and multithreaded algorithm have tested 28 DNA sequences using one and two processors machines. The comparison results show that the multithreaded version performs better than single threaded version. For a multithreaded version on a single processor, a performance improvement of 36 % is realized compared to a single threaded version. On a two processor machine, the multithreaded version performance improvement was found to be 40% compared to single threaded version. The parallel (multithreaded) Damerau Levenhstein distance algorithm achieved the reduction of the execution time because of workload distribution among different threads. Each thread worked on its sequences to perform comparison and calculate the similarity scoring value. The word multithreading can be translated as many threads of control. While a traditional process always contains a single thread of control, multithreading separates a process into many execution threads, each of which runs independently.The improved speedup is attributed to the benefits that arise from multithreading which are:

- Improved application responsiveness and better program structure - any program in which many activities do not depend upon each other can be redesigned so that each activity is executed as a thread.
- Efficient use of multiple processors - The algorithm can run much faster when implemented with threads on a multiprocessor.
- Use fewer system resources - The cost of creating and maintaining threads is much smaller than the cost for processes, both in system resources and time.

Scalability is a concern with bio-sequence analysis. As volumes of data increase, it often becomes necessary to distribute the workload over several threads (software parallelism) or processors (Hardware parallelism).

5.2.2 Experimental results using different number of threads

Experimental results are shown for a number of threads and compared with single-threaded results (speedup). Measurements were made on a low loaded system with no other users logged on.

The algorithm speedup is reached only for sample sizes greater than 200. The reason for this is the synchronization time between threads, especially at the end of each step, when all threads must be waited, and then started again.The results also show that speedup decreases as the number of threads increases over the number of available processors. The optimum is reached when the number of threads is equal to the number of processors.

Based on the results presented above, we conclude that under certain conditions the multithreading can improve the performance of the algorithm which are running on multiprocessor system. If threads run independently or with very low communication, speedup is only limited by the number of processors. If the communication between threads is heavier, speedup can be reached only if the time of computation between synchronization is at least several times greater than the synchronization time.

5.3 Recomendations

This project in an attempt to provide a solution to the above problems, we implemented a multi-threaded version of Damerau Levenhstein algorithm. This algorithm was chosen due to its appropriate nature to model biological mutations which are inherently common in biological sequences. By parallelizing this algorithm using multithreaded programming model, its speed is improved significantly.

Summarizing the developments in the parallelized Damerau Levenhstein distance algorithm, we can state the major speed increase is the result of multithreading. Currently the main development direction of processor manufacturers is the increase in the number of cores (processors), so a parallelized version of this algorithm would be able to take even better advantage of this trend in the future. We recommend a further research on this by testing on more processors to ascertain its performance.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 Conclusion

From the results presented above, we can make a general conclusion that under certain conditions; multi-threading programming can improves the performance of Damerau Levenshstein distance algorithm. When threads run independently or with very low communication overhead, speedup is only limited by the number of processors. If the communication overheads between threads are huge, speedup can be reached only if the computation time between synchronization is greater than the synchronization time. When we observed the speedup using different number of threads, we observed that the speedup slowly decreases as load increases, which is the result of a slower communication overhead through the operating system.

Algorithms play an important role in bioinformatics and computational biology. This helps in analysis of bio-sequences and thus facilitates the understanding of biological processes through the application of statistical and heuristic techniques. Much of the previous attempts have focused on sequence alignment algorithms, by implementing Damerau Levenshtein distance algorithm and parallelizing it using multithreading technology will usher in fresh insights in designing sequence comparison algorithms using software parallelism (multithreading).

This prototype implementation of the Damerau Levenshtein distance algorithm demonstrates that the algorithm can be parallelized for fast operation in searching sequence databases and validates the usage of multithreading to increase the speed of the algorithm (Damerau Levenshtein distance).This algorithm efficiently distributes the patterns to be searched on multiple threads to achieve rapid sequence matching operation. The algorithm is designed to fully exploit thread level parallelism to enhance searching speed. By distributing a large number of patterns over multiple threads, the algorithm shows better performance. From detailed experiments and performance analysis, our algorithm shows remarkable performance gain compared to the original Damerau Levenhstein algorithm. Our method provides a high performance multithreaded approach which exploits both instruction level and data level parallelisms. On conclusion our approach achieves performance enhancement through not only multi-threading but also data decomposition. Summarizing the findings, we can conclude that the major speed increase is the result of multithreading. Nowadays the main development direction of processor manufacturer is the increase in the number of core processors, so multithreading would be able to take even better advantage of this trend in the future. As a future work, we will extend the algorithm further targeting today's multi-core processors

6.2 Future Work

I propose a further study on the Damerau Levenshtein distance algorithm on today's multi-core architecture machines.

The prototype developed has the capability of using one and two cores CPUs but I could propose a further study on multiple cores (more than two cores) to measure the effect on performance. Although this may turn out to be a challenging study but it might give a new perspective on the study of parallelizing sequence comparison algorithms using multi-threading programming model on multiple cores.

The following are high level further improvements on the above algorithm implementation.

- Damerau Levenhstein algorithm can be enhanced by using more processors to speed sequence comparison (We have used up to two processors).
- Visualization of the output can be done by streaming the outputs to available visualization tools.
- Statistical analysis and comparison of the results generated between Damerau Levenhstein algorithm and sequence alignment algorithms on the same dataset can be done to ascertain the optimal one between the two.

REFERENCES

1. Hunt, E Atkinson, MP &Irving RW 2001, 'A *Database Index to Large Biological Sequences'*, Department of Computing Science, University of Glasgow, Proceedings of the 27th VLDB Conference, Italy.
2. Park, JH & George, KM 1999, '*Efficient parallel hardware algorithms for string matching', Microprocessors and Microsystems,* vol. 23, pp. 155-168, USA
3. Gusfield, D 1997, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.*: Cambridge University Press, New York, NY
4. Needleman, S.B. & Wunsch, C.D 1970. *'A general method applicable to the search for similarities in the amino acid sequence of two proteins'.* Journal of Molecular Biology, 48(3), 443-53.
5. Damerau, F.J 1964, '*A technique for computer detection and correction of spelling error's*, Communications of the ACM
6. Baxevanis, DA & Ouellette, Francis, BF 2005, '*Bioinformatics: A practical guide to the analysis of genes and proteins'*, New Jersey: John Wiley & Sons Inc
7. Gribskov, M McLachlan, AD & Eisenberg, D 1987, '*Profile analysis: Detection of distantly related proteins'*, Proceedings of the National Academy of Sciences of the United States of America.
8. Bentley, DR 2006, '*Whole-genome re-sequencing. Current Opinion in Genetics & Development'* Solexa Ltd, Chesterford Research Park, Little Chesterford, Near Saffron Walden, Essex, CB10 1XL, UK
9. Lipson, A & Hazelhurst, S 2001, '*DNA Pattern matching using FPGAs*', Annual Pattern Recognition Association of South Africa Conference, 2001.
10. Houle, JL Cadigan, Henry, S Pinnamaneni, A Lundahl, S 2000, '*Database Mining in the Human Genome Initiative*', Whitepaper, Biodatabases.com, Amita Corporation. Available: http://www.biodatabases.com/whitepaper.html
11. Mahalingam, K & Bagasra, O 2003, '*Bioinformatics Tools: Searching for Markers in DNA/RNA Sequences'*,Claflin University, Orangeburg,SC, USA
12. Hunt, E, Malcolm, Atkinson, P & Robert, W.I 2004, '*A Database Index to Large Biological Sequence's,* Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, UK

13. Cao, X Li, SC Ooi, BC Tung AKH 2006, '*An Efficient Model for Similarity Search in DNA Sequence Databases*' Department of Computer Science, National University of Singapore, Singapore, 117543

14. Smith, TF & Waterman, MS 1981,*'Identification of common molecular subsequences'* J. Mol. Biol., 147 195-197.

15. Pepper, L 2001, '*Searching Sequence Databases'*, MPS Thesis

16. Wagner, R & Fischer, M 1974, '*The string-to-string correction problem*', Journal of the *ACM*

17. Durbin, R, Eddy, S Krogh & Mitchison, G 1998, 'Bilogical Sequence Analysis, Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge UK

18. Albrecht, FF 2001, '*An indexed and parallelized search engine for similar DNA sequences'*, Genoogle

19. Garey, MR & Johnson, 1990, '*Computers and Intractability: A Guide to the Theory of NP-Completeness'*, W. H. Freeman & Co. New York, NY, USA

20. Bare, JC 2005, '*Applied Algorithms'*, University of Washington, USA

21. Pepper, L 2001, '*Searching Sequence Databases*, MPS Thesis', Wells College, Aurora New York

22. Hughey, R 1993, '*Massively Parallel Biosequence Analysis*', *Technical Report UCSC-CRL-93-14*, University of California, Santa Cruz.

23. Lošo, MD 2010,'*Algorithms for efficient alignment-free sequence comparison'*

24. Bell, M 2009, '*Animating String Searching Algorithms'*, New Castle University

25. Knuth D., J. Morris, J. Pratt 1977, '*Fast Pattern Matching in Strings'*, *SIAM Journal on Computing*, Vol.6, No.2, pp.323–350.

26. Xao, C 2006, '*Approximate Matching in Genomic Sequence Data'*, National University Of Singapore

27. Pappas, NP 2003, *'Searching Biological Sequence Databases Using Distributed Adaptive Computing'*,Bradley Blacksburg, Virginia, 2003

28. Surendirath, S 2005, '*Accelerating DNA sequential analysis through exploiting parallel hardware and reconfigurable computing*', The University of Cincinnati, India

29. Holloway, JL 1992, '*Algorithms for String Matching with Applications in Molecular Biology*', Oregon State University,USA

30. Sheik, SS Aggarwal, SK Poddar, A. & Sekar, A 2005, '*Analysis of string-searching algorithms on biological sequence databases*', Bioinformatics Centre and Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India, 2005

31. Johnstone, J 2008, '*A survey of sequence matching and alignment algorithms*',2008

32. Mount, DW 2004, '*Bioinformatics Sequence and Genome Analysis*',Cold spring Harbor Labaroty press, New Yor USA

33. Kattamuri, KR 2003, '*Algorithms for searching a Beowulf cluster*', Melbourne Florida,USA

34. Koonin, G 1999, '*The emerging paradigm and open problems in comparative genomics*'. Bioinformatics,National Center for Biotechnology, USA

35. Pabbaraju, S 2007, '*A DNA sequence assembly program that processes genetic sequences to produce high quality counting sequences*'

36. Russell, A & Hogg J 2001, '*Biological Sequence Analysis*', University Of Washington,USA

37. Karp, SS & Rabin MO 1987, '*Efficient randomized pattern matching algorithms*' IBM Journal of Research and Development, 32:249 260, USA

38. Berg, M & Sarha, T 2004, '*Performance Comparison of String Search Algorithms*',Finland

39. Anitha, V & Poorna, B 2005, '*Improved Algorithm for Global Alignment in DNA sequencing*'

40. Kabir, M 2009, '*Similarity Matching techniques for fault diagnosis in electronics*',University of Tuebingen , Germany

41. Kukich, K 1992, '*Techniques for automatically correcting words in text*' ACM Computing Surveys,USA

42. Cormen, TH 2000 *'Introduction to Algorithms'*, The MIT Press, Cambridge, Massachusetts London, England

43. Levenshtein, VI 1965, '*Binary codes capable of correcting spurious insertions and deletions of ones'*, volume 1. 1965.18.

44. Smith F &. Waterman, MS 1981, '*Comparison of bio-sequences', Adv. Appl.* 482-489

45. Gotoh, O 1982, 'An improved algorithm for matching biological sequences' *Journal of Molecular Biology*, vol. 162, pp. 705–708

46. Cormen, TH et al 1989, '*Introduction to Algorithms'*, The MIT Press,Cambridge, Massachusetts London, England

47. Primrose, SB 1998, '*Principles of Genome Analysis: a guide to mapping and sequencing DNA from different organisms.'* 2nd Ed. 1998. Blackwell Science: Oxford. ISBN 0-632-04983-9.

48. Katam, S 2002, ' *A Scalable Architecture for High Speed DNA Pattern Matching'*,Master's thesis, University of Cincinnati, Finland

49. Krawetz AS & Womble Dd 2003, ' *Introduction to Bioinformatics'*, New Jersey: Humana Press Inc,USA

50. Altschul, S.F. *The Statistics of Sequence Similarity Scores*,http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html

51. Myers, E 1991 '*An overview of sequence comparison algorithms in molecular biology'*, Tech. Rep. TR-91-29, Dept. of Computer Science, University of Arizona.

52. Krane, D.E & Ramyer, M.L 2003 *'Fundamental Concepts of Bioinformatics'*, Pearson Education

53. Gusfield D 1999, *'Algorithms on strings , Trees and sequences'*, Computer Science and Computational Biology, Cambridge University Press, UK

54. String searching Algorithm,en.wikipedia.org/wiki/String_searching_algorithm[Online;accessed 6th March 2011]

55. Editdistance,http://www.cs.mcgill.ca/~adubra/teaching/comp202/edit_distance .pdf[Online;accessed 7-April-2011].

56. Needleman-wunsch algorithm, http://en.wikipedia.org/w/index.php?title=Needleman-Wunsch_algorithm&oldid= 116968647, 2007. [Online; accessed 6-May-2011].

57. Smith-waterman algorithm, //en.wikipedia.org/w/index.php?title=Smith-Waterman_algorithm&oldid=126482240,2011. [Online; accessed 16th-June-2011].

58. Needleman-wunsch algorithm, http://www.maths.tcd.ie/~lily/pres2/sld009.htm. [Online; accessed 4th-April-2011].

59. Stuart M. Brown. Needleman-wunsch algorithm, http://www.med.nyu.edu/rcr/rcr/course/sim-sw.html. [Online; accessed 4th -April-2011].

60. Paul E.Black. Smith-waterman algorithm, http://www.nist.gov/dads/HTML/smithWaterman.html, 2006. [Online; accessed 12th-March-2011].

61. M.A. Kentie, Biological Sequence Alignment Using Graphics Processing Units http://kentie.net/article/thesis/thesis.pdf [Online; accessed April-2011]

62. Anderson, R., Francis, B., Homer, A., Howard, R., Sussman, D. and Watson. (2001) *Professional ASP.NET.* Wrox Press Ltd.

APPENDIXES

1.1 Damerau Levenshtein algorithm

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace DNAProject
{
   public class DamerauLevenshtein
   {
      private char[] compOne;
      private char[] compTwo;
      private int[,] matrix;
      private Boolean calculated = false;

      public DamerauLevenshtein(String a, String b)
      {
         if ((a.Length > 0) || (b.Length > 0))
         {
            compOne = a.ToCharArray();
            compTwo = b.ToCharArray();
         }
      }

      public int[,] getMatrix()
      {
         setupMatrix();
         return matrix;
      }

      public int getSimilarity()
      {
         if (!calculated) setupMatrix();

         return matrix[compOne.Length, compTwo.Length];
      }

      private void setupMatrix()
      {
         int cost = -1;
         int del, sub, ins;

         matrix = new int[compOne.Length + 1, compTwo.Length + 1];

         for (int i = 0; i <= compOne.Length; i++)
         {
            matrix[i, 0] = i;
         }

         for (int i = 0; i <= compTwo.Length; i++)
         {
            matrix[0, i] = i;
         }
         for (int i = 1; i <= compOne.Length; i++)
         {
            for (int j = 1; j <= compTwo.Length; j++)
```

```
        {
            if (compOne[i - 1] == compTwo[j - 1])
            {
                cost = 0;
            }
            else
            {
                cost = 1;
            }

            del = matrix[i - 1, j] + 1;
            ins = matrix[i, j - 1] + 1;
            sub = matrix[i - 1, j - 1] + cost;
            matrix[i, j] = minimum(del, ins, sub);

            if ((i > 1) && (j > 1) && (compOne[i - 1] == compTwo[j - 2]) && (compOne[i - 2] == compTwo[j
- 1]))
            {
                matrix[i, j] = minimum(matrix[i, j], matrix[i - 2, j - 2] + cost);
            }
        }
    }

    calculated = true;
    //PrintMatrix(matrix, "###0", @"c:\dumpt.txt");
    //displayMatrix();
}

private void displayMatrix()
{
    Console.WriteLine("  " + compOne);
    for (int y = 0; y <= compTwo.Length; y++)
    {
        if (y - 1 < 0) Console.Write(" "); else Console.Write(compTwo[y - 1]);
        for (int x = 0; x <= compOne.Length; x++)
        {
            Console.Write(matrix[x, y]);
        }
        Console.WriteLine();
    }
}

private int minimum(int d, int i, int s)
{
    int m = int.MaxValue;
    if (d < m) m = d;
    if (i < m) m = i;
    if (s < m) m = s;

    return m;
}

private int minimum(int d, int t)
{
    int m = int.MaxValue;

    if (d < m) m = d;
    if (t < m) m = t;

    return m;
}
```

```
    private void PrintMatrix(int[,] M, string format, string path)
    {
       string row = "";

      row = "  " + compOne + "\n";
      DumpToFile(path, new string[] { row });
      row = "";
       for (int y = 0; y <= compTwo.Length; y++)
       {
          row += "[";
          if (y - 1 < 0)
             row += " ";
          else
             row += String.Format("{0, 8:c}", compTwo[y - 1].ToString());
          for (int x = 0; x <= compOne.Length; x++)
          {
             row += String.Format("{0, 8:c}", matrix[x, y].ToString());
          }
          row += "]";
          DumpToFile(path, new string[] { row });
          row = "";
       }
       return;
    }
    private void DumpToFile(string path, string[] linesToWrite)
    {
       using (StreamWriter sw = File.AppendText(path))
       {
          //File.WriteAllLines(path, linesToWrite );
          foreach (string text in linesToWrite)
             sw.WriteLine(text);
       }
    }
  }
}
```

1.2 Bio-sequence search engine

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using dnase;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
using System.Collections.Concurrent;


namespace DNAProject
{
  public class PEI
  {
    int tP;

    public int TP
    {
      get { return tP; }
      set { tP = value; }
    }
    string nM;

    public string NM
    {
      get { return nM; }
      set { nM = value; }
    }
    int x;

    public int X
    {
      get { return x; }
      set { x = value; }
    }
    long y;

    public long Y
    {
      get { return y; }
      set { y = value; }
    }
    int tC;

    public int TC
    {
      get { return tC; }
      set { tC = value; }
    }

  }
  public class KVP
  {
    string key;

    public string Key
    {
        get { return key; }
        set { key = value; }
      }
      int value;

      public int Value
      {
        get { return this.value; }
        set { this.value = value; }
      }
    }
    public class dnase
    {
      public List<KVP> GetPerc()
      {
        List<KVP> lst = new List<KVP>();
        for (int i = 10; i < 100; i++)
        {
          lst.Add(new KVP
          {
            Value = i
          });

        }
        return lst;
      }
      public List<PEI> Perf(string segment, int percentage)
      {
        List<PEI> lst = new List<PEI>();
        using (DNADataClassesDataContext db = new DNADataClassesDataContext())
        {
          int count = db.DNATables.Count();
          long ElapsedMilliseconds;
          int px = 0;
          while( true)//px < count )//for (int i = 1; i <= 10; i++)
          {
            px = (count >= 100 ? ( px < 1000 ? px + 100 : px + 500 ) : count);
            if (px > count) break;
            SearchSequential(segment, percentage, px, db, out ElapsedMilliseconds);
            lst.Add(new PEI
            {
              NM = "SERIAL",
              X = px,
              Y = ElapsedMilliseconds
            });
            int threads = SearchParallel(segment, percentage, px, db, out
ElapsedMilliseconds).Select(x=>x.ThreadID).Distinct().Count();
            lst.Add(new PEI
            {
              NM = "PARALLEL",
              X = px,
              Y = ElapsedMilliseconds,
              TC = threads
            });
          }

        }
        return lst;
      }
      public List<DNAI> Search(string segment, int percentage, int algorithm, out long ElapsedMilliseconds)
      {
```

```
    using (DNADataClassesDataContext db = new DNADataClassesDataContext())
      {
        int count = db.DNATables.Count();
        if (algorithm == 0)
          return SearchSequential(segment, percentage, count, db, out ElapsedMilliseconds);
        else if (algorithm == 1)
          return SearchParallel(segment, percentage, count, db, out ElapsedMilliseconds);
        else
          return SearchSequential(segment, percentage, count, db, out ElapsedMilliseconds);
      }

    }

    private List<DNAI> SearchSequential(string segment, int percentage, int count,
DNADataClassesDataContext db, out long ElapsedMilliseconds)
    {
      segment = segment.Trim().ToUpper();
      if (segment.Length == 0)
        throw (new Exception("Search string parameter is required."));

      Stopwatch sp = new Stopwatch();
      sp.Start();
      List<DNAI> lst = new List<DNAI>();

      int sim = int.MinValue;
      int lng = 0;
      int mtc = 0;
      var dQ = from d in db.DNATables.Take(count)
            select d;
      foreach (var dna in dQ)
      {
        DamerauLevenshtein dl = new DamerauLevenshtein(segment, dna.sequence);
        sim = dl.getSimilarity();
        lng = dna.sequence.Length;
        mtc = lng - sim;

        if ((mtc * 100) / lng >= percentage)
        {
          lst.Add(new DNAI
          {
            Id = sim,
            Name = dna.Name,
            Definition = dna.Definition,
            Sequence = dna.sequence
          });
        }
      }

      sp.Stop();
      ElapsedMilliseconds = sp.ElapsedMilliseconds;
      if (lst.Any())
        lst = (from l in lst
            orderby l.Id
            select l).ToList();
      return lst;
    }
    private List<DNAI> SearchParallel(string segment, int percentage, int count, DNADataClassesDataContext
db, out long ElapsedMilliseconds)
    {
      segment = segment.Trim().ToUpper();
      if (segment.Length == 0)
```

```
            throw (new Exception("Search string parameter is required."));

        Stopwatch sp = new Stopwatch();
        sp.Start();
        List<DNAI> lst = new List<DNAI>();

        var dQ = from d in db.DNATables.Take(count)
                 select d;
        object monitor = new object();

        Parallel.ForEach<DNATable, List<DNAI>>(dQ, () => new List<DNAI>(), (dna, loop, list) =>
        {
            int sim = int.MinValue;
            int lng = 0;
            int mtc = 0;
            DamerauLevenshtein dl = new DamerauLevenshtein(segment, dna.sequence);
            sim = dl.getSimilarity();
            lng = dna.sequence.Length;
            mtc = lng - sim;
            if ((mtc * 100) / lng >= percentage)
            {

                list.Add(new DNAI
                {
                    Id = sim,
                    Name = dna.Name,
                    Definition = dna.Definition,
                    Sequence = dna.sequence,
                    ThreadID = System.Threading.Thread.CurrentThread.ManagedThreadId
                });

            }
            return list;
        },
        (finalResult) => { lock (monitor)lst.AddRange(finalResult); });
       sp.Stop();
        ElapsedMilliseconds = sp.ElapsedMilliseconds;
        if (lst.Any())
            lst = (from l in lst
                   orderby l.Id
                   select l).ToList();
        return lst;
    }

  }

}
```

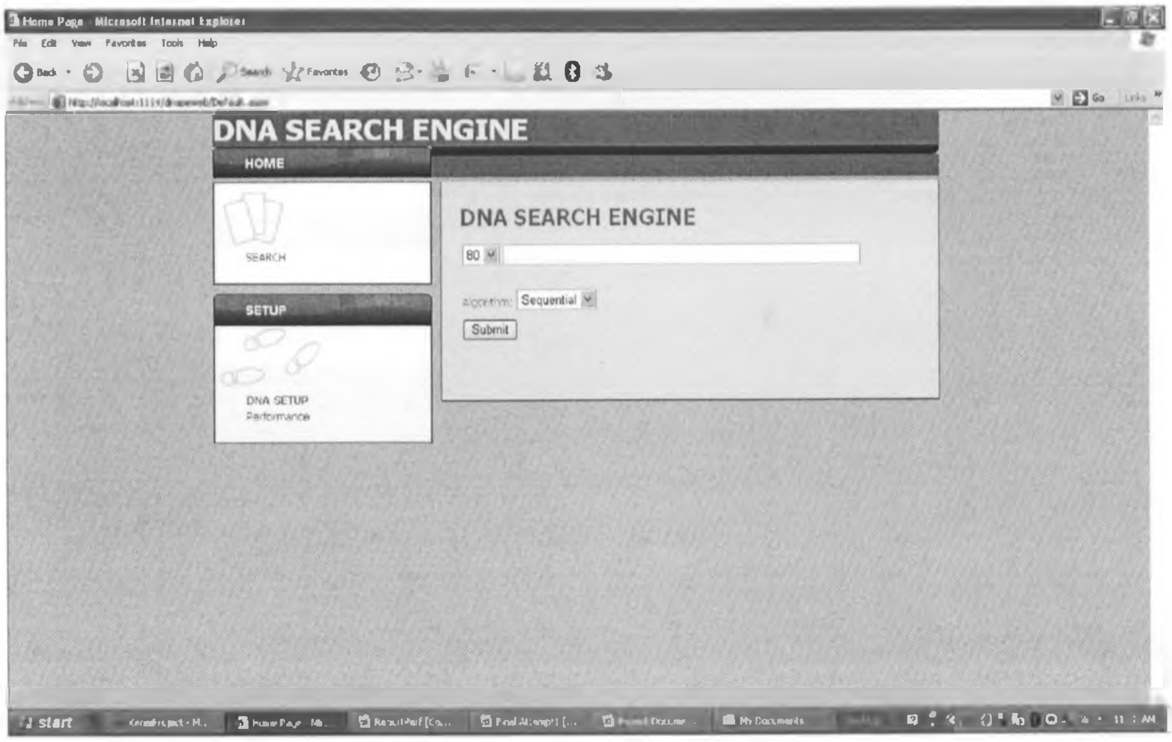
1.3 Database connection

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using dnase;

namespace KeneiProject
{

  public class Database
  {
    public static bool DNAValid(string dna)
  {
    dna = dna.Trim().ToUpper();
    if (dna.Length == 0) return false;

    var vQ = from v in dna.ToCharArray()
           where v != 'T' && v != 'G' && v != 'A' && v != 'C'
           select v;
    if (vQ.Any())
      return false;
    else
      return true;
  }

  public static DNATable GetDNA(string name)
  {
    name = name.Trim();
    if(name.Length == 0 )
      throw( new Exception("DNA Name parameter is required."));

    DNATable rv = new DNATable();
    using (DNADataClassesDataContext db = new DNADataClassesDataContext())
    {
      var dQ = from d in db.DNATables
             where d.Name == name
             select d;
      if (!dQ.Any())
        throw (new Exception("Given DNA identifier '" + name + "' does not exists."));
      rv = dQ.First();
    }
    return rv;


  }

  public void DNAInsert(DNATable dna)
  {
    dna.Name = dna.Name.Trim();
    if (DNAValid(dna.sequence))
      throw( new Exception("DNA sequence contains invalid characters."));
    using (DNADataClassesDataContext db = new DNADataClassesDataContext())
    {
      var dQ = from d in db.DNATables
             where d.Name == dna.Name
             select d;
      if(dQ.Any() )
        throw( new Exception("Given DNA identifier '" + dna.Name + "' already exists."));

      db.DNATables.InsertOnSubmit(dna);
      db.SubmitChanges();
    }
  }
}
public class DNAI
{
  private int threadID;

  public int ThreadID
  {
    get { return threadID; }
    set { threadID = value; }
  }
```

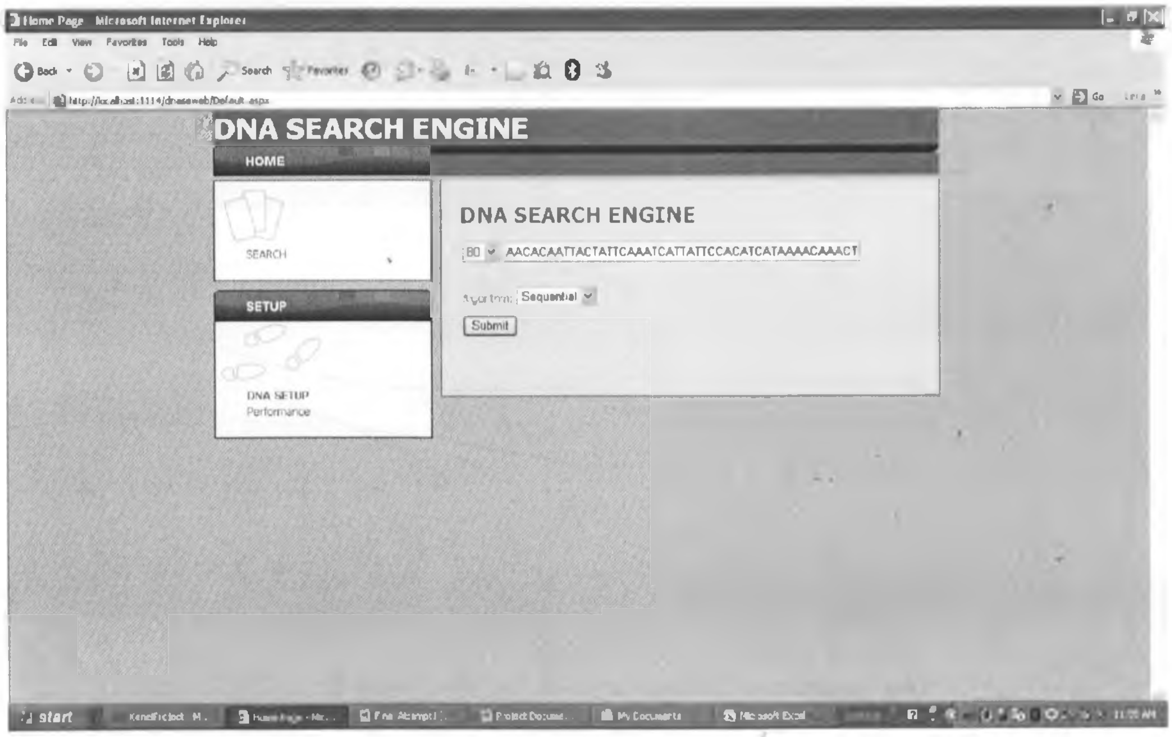
```
private int _id;

public int Id
{
   get { return _id; }
   set { _id = value; }
}

private string _Name;

public string Name
{
   get { return _Name; }
   set { _Name = value; }
}

private string _Definition;

public string Definition
{
   get { return _Definition; }
   set { _Definition = value; }
}

private string _AccessionNumber;

public string AccessionNumber
{
   get { return _AccessionNumber; }
   set { _AccessionNumber = value; }
}

private string _Sourceandtaxonomy;

public string Sourceandtaxonomy
{
   get { return _Sourceandtaxonomy; }
   set { _Sourceandtaxonomy = value; }
}

private string _CompleteliteratureReferences;

public string CompleteliteratureReferences
{
   get { return _CompleteliteratureReferences; }
   set { _CompleteliteratureReferences = value; }
}

private string _Commentsandkeywords;

public string Commentsandkeywords
{
   get { return _Commentsandkeywords; }
   set { _Commentsandkeywords = value; }
}

private string _FEATURE;

public string FEATURE
{
   get { return _FEATURE; }
```

```
            set { _FEATURE = value; }
        }

        private string _summary;

        public string Summary
        {
            get { return _summary; }
            set { _summary = value; }
        }

        private string _sequence;

        public string Sequence
        {
            get { return _sequence; }
            set { _sequence = value; }
        }
    }
}
```

1.4 Sample screens for Bio-sequence search engine

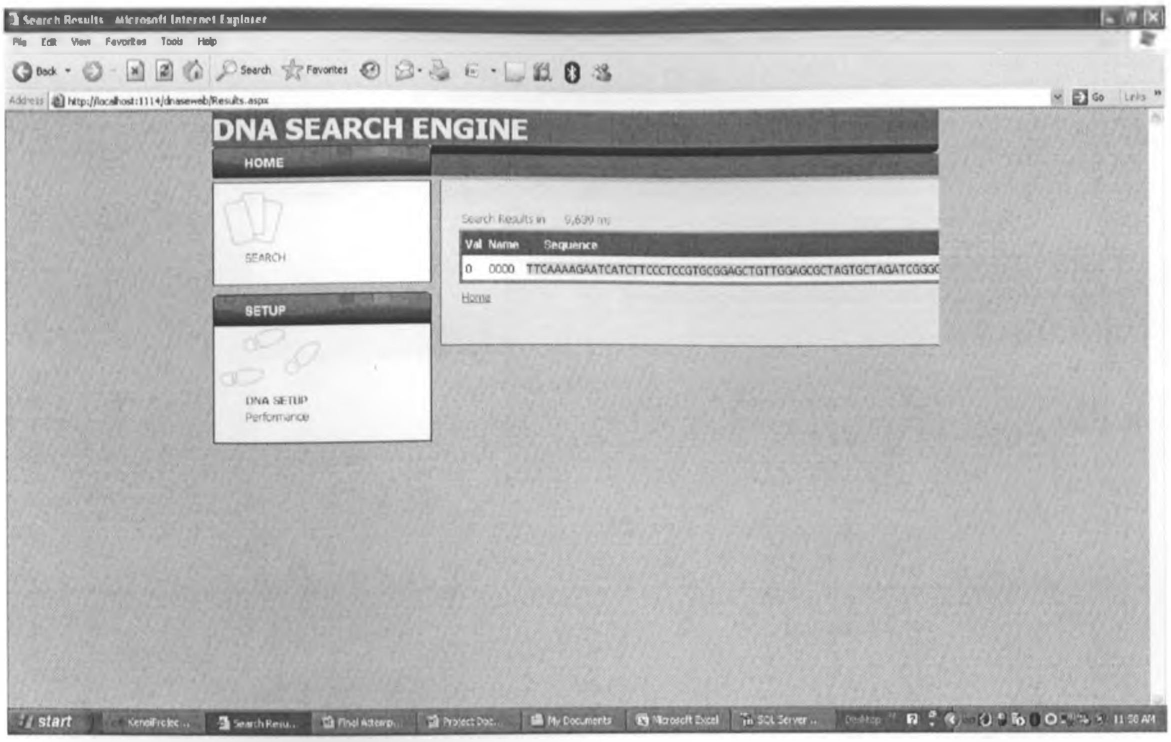


1.5 Sequential Search


Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
http://localhost:1114/dnaseweb/Default.aspx
DNA SEARCH ENGINE
HOME
SEARCH
SETUP
DNA SETUP
Performance
DNA SEARCH ENGINE
80
Algorithm: Sequential
Submit
start

80 AACACAATTACTATTCAAATCATTATTCCACATCATAAAACAAACT

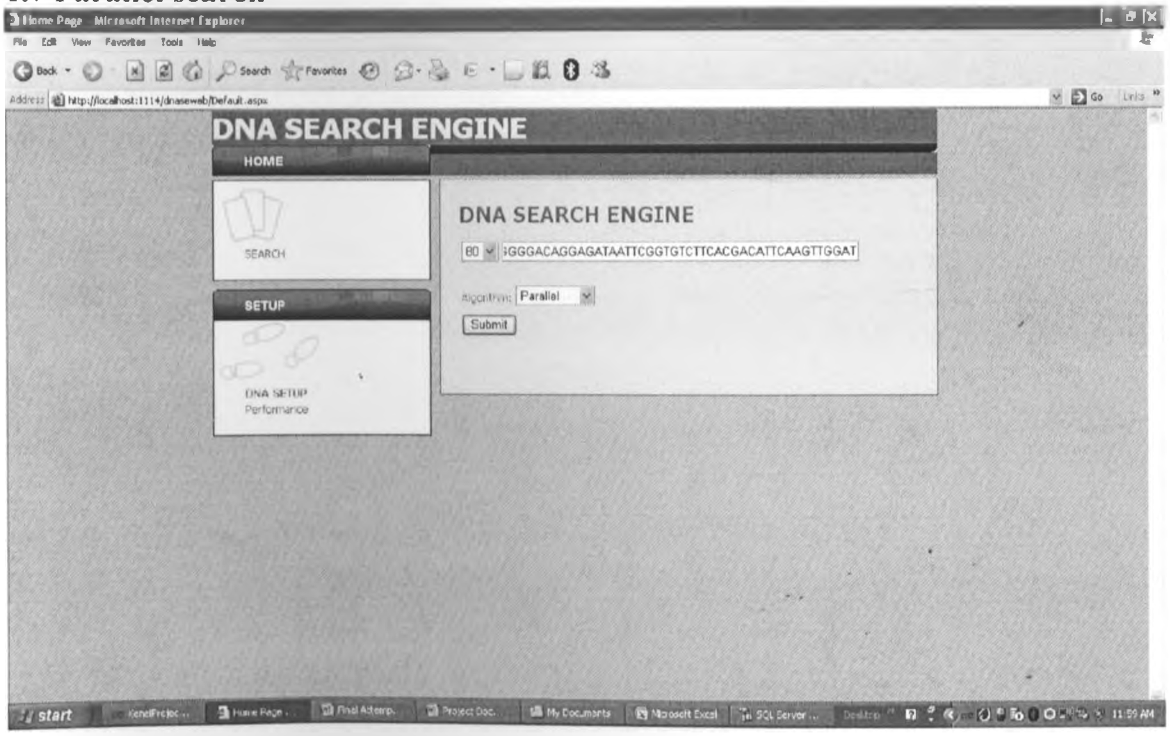
1.6 Sequential search results


Search Results in 9,639 ms

Val	Name	Sequence
0	0000	TTCAAAAGAATCATCTTCCCTCCGTGCGGAGCTGTTGGAGCGCTAGTGCTAGATCGGGC

Search Results - Microsoft Internet Explorer
http://localhost:1114/dnaseweb/Results.aspx
DNA SEARCH ENGINE
HOME
SEARCH
SETUP
DNA SETUP
Performance
Home

1.7 Parallel search

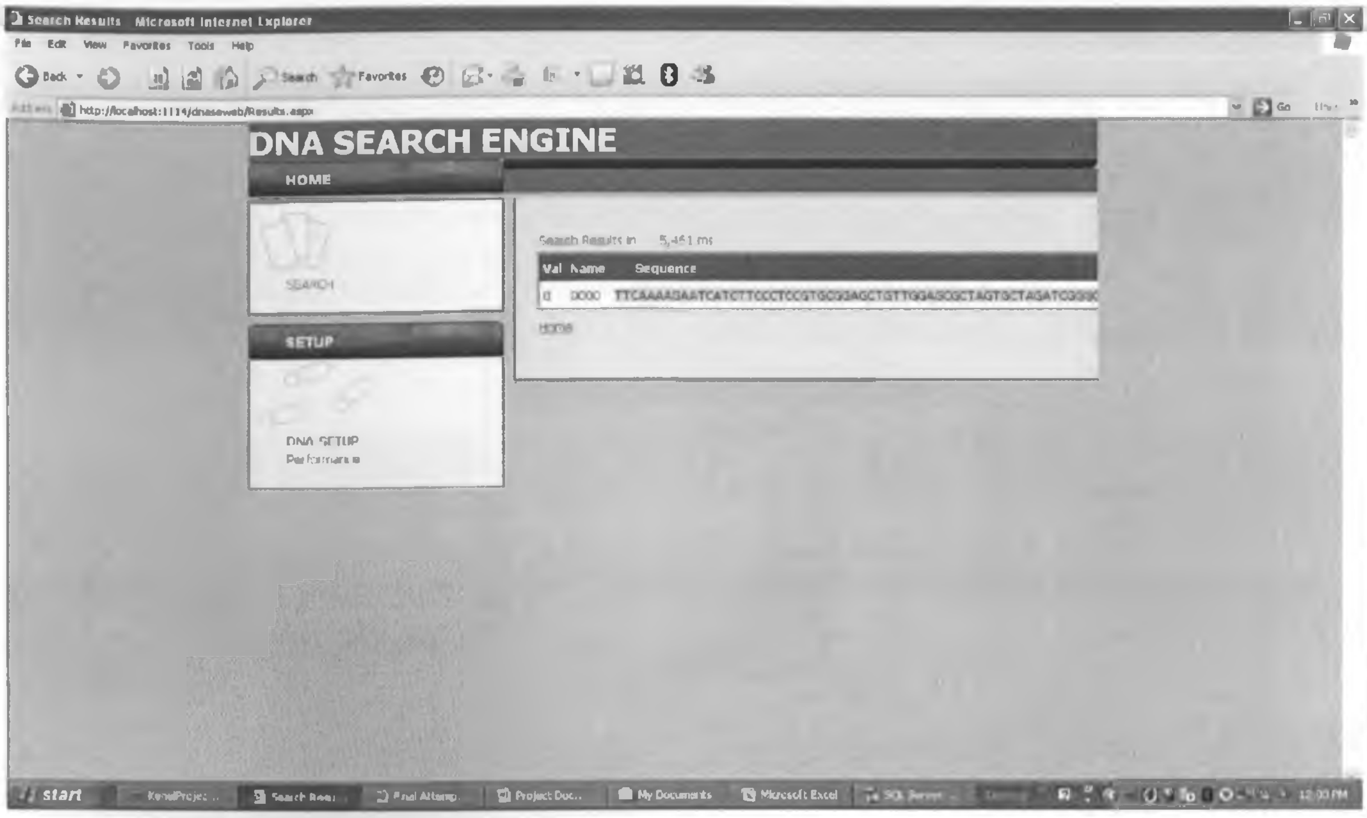

DNA SEARCH ENGINE

80 GGGACAGGAGATAATTCGGTGTCTTCACGACATTCAAGTTGGAT

Algorithm: Parallel

Home Page - Microsoft Internet Explorer
http://localhost:1114/dnaseweb/Default.aspx
DNA SEARCH ENGINE
HOME
SEARCH
SETUP
DNA SETUP
Performance
Submit

1.8 Parallel search results

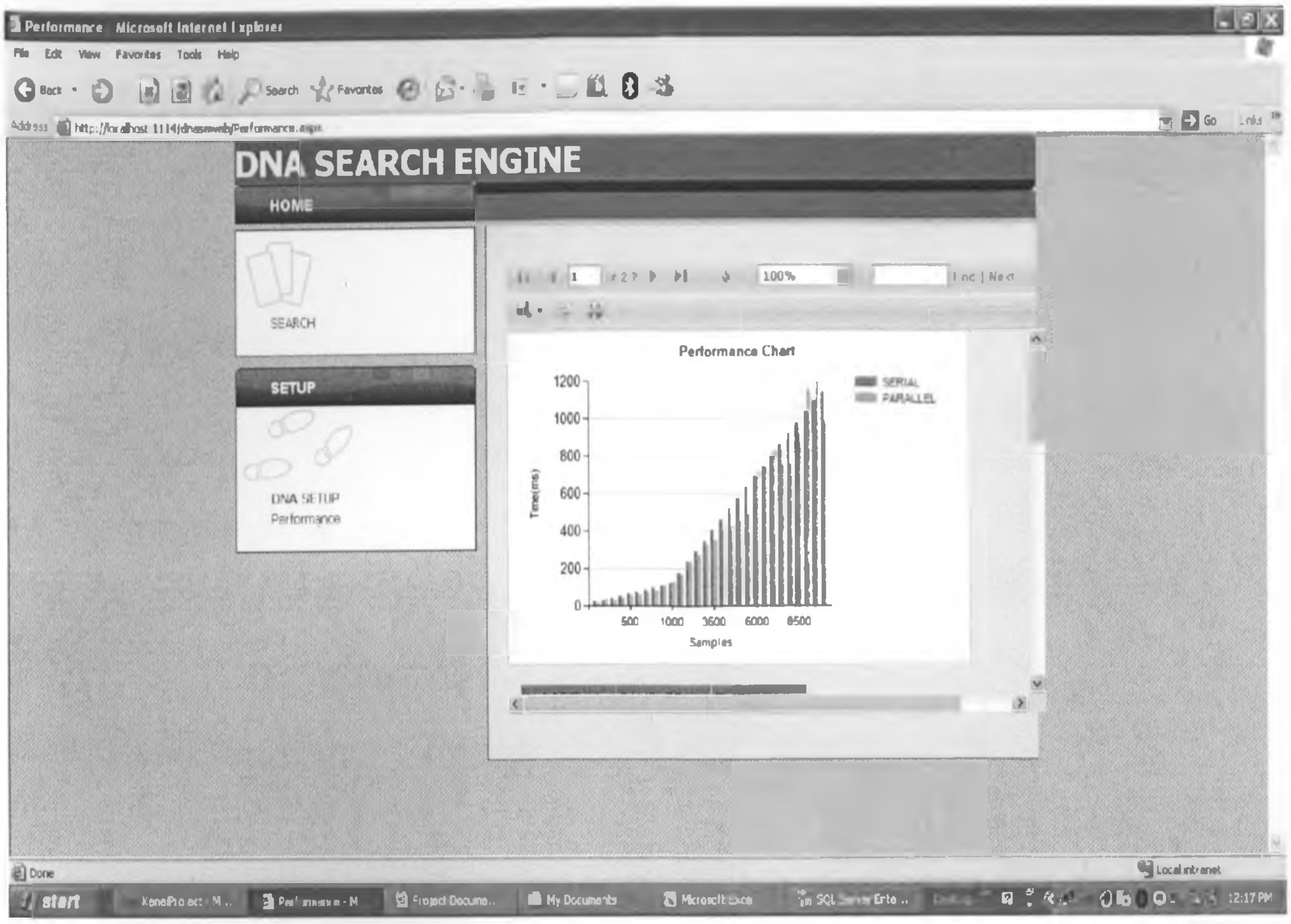

Search Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
http://localhost:1114/dnaseweb/Results.aspx
DNA SEARCH ENGINE
HOME
SEARCH
SETUP
DNA SETUP
Performance
start

Search Results in 5,461 ms

Val	Name	Sequence
0	0000	TTCAAAAGAATCATCTTCCCTCCGTGCGGAGCTGTTGGAGCGCTAGTGCTAGATCGG

Home

1.9 Performance comparisons


Performance - Microsoft Internet Explorer
DNA SEARCH ENGINE
HOME
SEARCH
SETUP
DNA SETUP
Performance
Performance Chart
SERIAL
PARALLEL
Time(ms)
Samples
0
200
400
600
800
1000
1200
500
1000
3500
6000
8500