A MULTIPROCESSOR PARALLEL APPROACH TO BIT-PARALLEL APPROXIMATE STRING MATCHING

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Elias Anwar Chibli

March 2008

A MULTIPROCESSOR PARALLEL APPROACH TO BIT-PARALLEL APPROXIMATE STRING MATCHING

A Thesis

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Elias Anwar Chibli

March 2008

Approved by:

Keith Evan Schubert, Advisor, Computer Science	Date
Ernesto Gomez	
Yasha Karant	

ABSTRACT

Efficient approximate string matching algorithms are an essential part of the growing field of Bioinformatics. The search to uncover the meaning of DNA sequences has quickly increased in the past few decades. The demand for low latency string matching systems is a must as the size of data sets involved is ever increasing. A parallel architecture of these search mechanisms provides an efficient method for minimizing latency. The technique proposed applies two methods of parallelization; a bit-parallel approach and also the use of multiple processors in parallel via a cluster of workstations using MPI. Empirical results show that applying a parallel design to string matching algorithms is a viable solution. For a system of between two and eight heterogeneous workstations, the results produce nearly ideal speedup for most practical cases.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Schubert who has helped me a great deal in this process and has been a mentor throughout my computer science education. He has inspired me with his attitude and thirst for knowledge and truth of all things. His advice was crucial in enabling me to pursue this path. He helped me believe that this was a goal I could achieve.

I also want to thank Dr. Karant and Dr. Gomez for providing their knowledge, advice and technical support during this project. I also want to extend my thanks to all professors in the computer science department at CSUSB. Each one of you has helped me in many ways to better understand this field. It was truly a pleasure being around you both in and out of the classroom.

I wish to give my sincere thanks to my family. I am grateful for my wife, son and daughter who have always supported me through it all in the many years it has taken me to complete this goal. I love you all and I am so proud of you. I also want to thank my mom and dad who are my heroes. You are both truly special human beings with a beautiful spirit. I would not be the person I am today without your continued guidance. I also want to thank my brother who has inspired me with his own achievements and way of life. Your moral support and understanding is priceless. I am grateful just to have you all in my life.

DEDICATION

To Allison, Elias and Aliyah.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

This thesis aims to present with empirical results that a parallel design with the use of multiple processors can be successfully applied along with bit-parallel approximate string matching algorithms to solve practical Bioinformatics problems. It will demonstrate that nearly optimal speedup can be achieved with a cluster of between two and eight workstations using MPI(Message Passing Interface), directly decreasing the total latency required to perform a string matching problem. This introductory chapter examines the background of string matching algorithms in Section 1.1. An overview of the field of Bioinformatics is explained in Section 1.2. Finally, section 1.3 introduces some practical Bioinformatics problems and how they can be solved using string matching solutions.

1.1 String Matching Background

The problem of matching strings is one that is deeply rooted in the field of computer science. It has been studied to a great extent since the early beginnings of modern digital computation. In its essence, the problem of matching strings is one that is very general. It is the problem of verifying how similar or different two sets of data are from one another. The metric used to measure the result of the comparison can vary according to the given application. There is a vast variety of applications that make

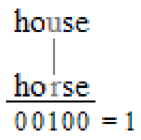


Fig. 1.1: Hamming Distance Example

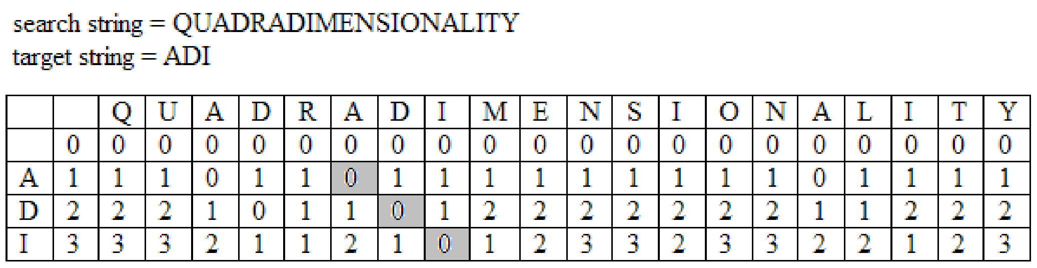
use of string matching; such as database searching, spell checkers in word processors, web searching, anti-virus software, document comparison tools, DNA sequencing, and many more. These and many other common computing tools need accurate and efficient string matching algorithms in order to operate. This makes string searching an significant and fundamental problem in computer science.

1.1.1 In The Beginning

There have been several algorithms proposed for solving string matching problems. In 1950, Richard W. Hamming published an article describing a method for detecting and correcting error codes[6]. Better known as the "Hamming distance", it is one of the earliest successful methods for computing the difference between two strings. This work is still used today in a variety of applications that span from telecommunications to cryptography. The Hamming distance was also used as a base for much of the subsequent research in string matching. Formally, the Hamming distance is defined as the number of positions containing differing characters in two aligned strings of equal size. This means that for binary strings α and β in the alphabet $\Sigma = \{0, 1\}$, the Hamming distance is the number of 1's of α xor β. Figure 1.1 illustrates an example of computing the Hamming distance. This approach to string matching is fast, efficient and sufficient for many applications, but has many restrictions. The two strings being compared are assumed to be perfectly aligned and therefore must be the same size. The resulting metric only tells us the amount of differing characters that exist between the two strings.

1.1.2 Levenshtein Distance

A more advanced technique for string matching was proposed by Vladimir I. Levenshtein in 1966[12]. He did not suggest an algorithmic solution, but merely the mathematical method. Known now as the "Levenshtein distance" or "edit distance", it allows for the matching of strings of different sizes. This brings about the advantage of being able to compare strings that are not directly aligned with one another. It also enables to check if a search string contains a subset similar to the target string more than once. In addition, it identifies not only direct mismatches, but differences due to additions, subtractions and substitutions of characters. Because of its flexibility, this method is the main root of most modern implementations of string matching algorithms for a variety of applications. The problem solved by the Levenshtein distance can be defined as follows: find all locations in a character search string t of finite length n that contain a substring similar to a target string p of finite length m where $n \geq m$. The similarity is defined by each subset having a maximum of k number of differences from the target string. It is assumed that both strings only contain characters that belong to a finite alphabet Σ. This, results in obtaining 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. The Lev-


search string = QUADRADIMENSIONALITY
target string = ADI

		Q	U	A	D	R	A	D	I	M	E	N	S	I	O	N	A	L	I	T	Y
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	0	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1
D	2	2	2	1	0	1	1	0	1	2	2	2	2	2	2	2	1	1	2	2	2
I	3	3	3	2	1	1	2	1	0	1	2	3	3	2	3	3	2	2	1	2	3

Fig. 1.2: Levenshtein Distance - Resulting Matrix

enshtein distance applies a dynamic programming approach that can be formulated as follows:

- Build an $(m+1)$ * $(n+1)$ matrix D, indexed as $D[i][j]$
- Initialize $D[*][0] = i$, $D[0][*] = 0$
- If $p[i] = t[j]$, $D[i][j] = D[i\text{-}1][j\text{-}1]$
- If $p[i] \neq t[j]$, $D[i][j] = 1 + \min(\ D[i\text{-}1][j], D[i\text{-}1][j\text{-}1], D[i][j\text{-}1])$
- Report all instances of $D[m][j] \leq k$

Where p = search string, n = size of search string, t = target string, m = size of target string, k = mismatch threshold.

The time complexity for this algorithm is $\Theta(nm)$ and the space complexity is also $\Theta(nm)$. Figure 1.2 shows an example of a resulting matrix from a complete search operation using the Levenshtein distance algorithm. The highlighted cells show the path to the exact match of the target string in a subset of the search string. The resulting cell values in the last row of the matrix indicate how different the target string is from the search string at that given subset in the search string. A resulting cell value of zero in the last row indicates an exact match. A value of one means that there is at least one change that must be applied for the subset or the target so they can be the same. A value of two means that there are two changes required and so on. A basic implementation of this algorithm in C++ using the STL is shown in Table 1.1.

Tab. 1.1: Levenshtein Distance ($\Theta(nm)$ Space)- C++ Code Sample

```
vector<int> LD_NM_Space( const std::string& source, const std::string& target, int k ){
  int sourceLen = (int)source.length();
  int targetLen = (int)target.length();
  vector<int> locationVector;//declare the location vector

  if( sourceLen == 0 || targetLen == 0 )//check for emptry strings
    return locationVector;

  //Build the matrix. It will be accessed as matrix[row, col]
  std::vector< std::vector<int> > matrix( targetLen + 1 );//sets the rows
  for( int i=0; i <= targetLen; i++)
    matrix[i].resize( sourceLen + 1 );//columns

  for( int i=0; i <= targetLen; i++ )//initialize the first column
    matrix[i][0] = i;

  for( int j=0; j <= sourceLen; j++ )//initialize the first row
    matrix[0][j] = 0;

  //perform the matching operation
  for( int i=1; i <= targetLen; i++ )//rows
  {
    for( int j=1; j <= sourceLen; j++ )//cols
    {
      if( target[i-1] == source[j-1] )
        matrix[i][j] = matrix[i-1][j-1];
      else
        matrix[i][j] = 1 + std::min( std::min( matrix[i-1][j-1], matrix[i-1][j] ), matrix[i][j-1] );

      if( i == targetLen && matrix[i][j] <= k )//record any matches <= k
        locationVector.push_back( j );
    }
  }

  return locationVector;
}
```

One can easily notice that the dynamic programming computation of each cell in the matrix only relies on data from previously computed neighbor cells to the left, top, and top-left. This in turn means that the algorithm can be reformulated and improved to only use $\Theta(2m)$ space. This is because all that is needed is the current and previously computed column in the dynamic matrix. Instead of maintaining all cells of the matrix for a total of $n*m$ cells, we may only maintain the current and last column of the dynamic matrix and during the process we can in a separate structure keep track of the location of the matches found. This simple improvement can greatly improve memory usage in the computing system when large strings are being matched. Table 1.2 shows a sample implementation of this method in C++ using the STL. Notice that these sample implementations take as input the complete search and target strings. This is possible for relatively small data sets. In a more realistic approach, the search data may be too large to maintain in memory at all times and so the algorithm must stream subsets of the data from file. This technique also improves the memory usage of the algorithm at any given time since it is working with smaller chunks of the total search string at one time.

Tab. 1.2: Levenshtein Distance ($\Theta(2m)$ Space) - C++ Code Sample

```
vector<int> LD_2M_Space( const std::string& source, const std::string& target, int k ){
  int sourceLen = (int)source.length();
  int targetLen = (int)target.length();
  vector<int> locationVector;//declare the location vector

  if( sourceLen == 0 || targetLen == 0 )//check for emptry strings
    return locationVector;

  //vector p will always be the current active vector.
  //vector q will always be "the previous vector"
  //the pointer *temp is needed to swap between the current and last vector
  int* p = new int[ targetLen + 1 ];
  int* q = new int[ targetLen + 1 ];
  int* temp;

  for( int j=0; j <= targetLen; j++ )//Initialize the first col
    q[j] = j;

  //perform the matching operation
  for( int i=1; i <= sourceLen; i++ )//cols
  {
    p[0] = 0;//first cell in the column is always zero

    for( int j=1; j <= targetLen; j++ )//rows
    {
      if( target[j-1] == source[i-1] )
        p[j] = q[j-1];
      else
        p[j] = 1 + std::min( std::min( q[j-1], q[j] ), p[j-1] );

      if( j == targetLen && p[j] <= k )//record any matches <= k
        locationVector.push_back( i );
    }

    //swap the active/previous vectors
    temp = p;
    p = q;
    q = temp;
  }

  delete[] p;
  delete[] q;
  return locationVector;
}
```

1.1.3 Improving Time Complexity

It is important to improve on the amount of space used by the algorithm, but in practice it is more important to improve on the time complexity of the Levenshtein distance algorithm. There has been extensive research in this area in the past decades, and great improvements to the overall time needed to execute the algorithm have been found. In 1985 Ukkonen[15] produced an algorithm capable of performing the match in an expected average time of $O(kn)$. In 1988 Landau and Vishkin[11] improved on this to assure a worse case time of $O(kn)$ using $O(n)$ space. In 1996, Wu, Manber and Myers[19] produced a search technique based on the 4-Russians algorithm that has an expected average execution time in $O(kn/log\ s)$ where $O(s)$ was to be dedicated time spent to access a lookup table. The success of some of these algorithms was based on the strategy to more intelligently segregate out regions of the search string in the dynamic matrix that would not produce matches within the threshold of k. Implementations that use this approach are referred to as "filtering" search algorithms. They were originally introduced in 1987 by Karp and Rabin[9] for searches involving exact matches, but the idea carried over into approximate string matching. These algorithms usually preprocess/parse the search string to eliminate all but those subsets that have a high probability of yielding the desired matches with the target string. The key in this method is to have a high level of "filtration efficiency", meaning that the algorithm can effectively remove a large percentage of the search regions that will not yield matches. In cases of high efficiency, these types of algorithms usually deliver the fastest results. The downfall to this technique is that as the mismatch threshold k increases in proportion to the target string m, the ability to filter out sections of the search string quickly diminishes to the point of no longer being effective at all.

1.1.4 The Bit-Parallel Approach

Begining in the early 1990's, researchers began to look for new ways to improve the performance of search algorithms by looking at ways to parallelize their execution. The idea was to minimize the time necessary to complete a search operation by doing multiple computations at the same time instead of having the algorithm execute in a fully serialized mode. After vast analysis into the nature of the computations that are necessary to generate the dynamic matrix in the Levenshtein distance algorithm, a new idea was born; a paradigm known as "bit-parallel" or "bit-vector" search algorithms. These algorithms take advantage of the fact that modern computer systems perform computations on vectors of binary data (i.e., a machine word). The idea is to organize several pieces of information into a single bit-vector such that each will be computed in a single operation instead of doing each of those pieces separately in multiple computations. This ultimately provides a method of parallelization with a maximum possibility of gaining a degree of parallelism of w, given that a single bit is used to represent the needed information during computation, where w is the word size of the machine. What makes this approach so unique is that unlike most methods of parallelization, it does not depend on multiple processing units. The parallelization occurs with a single processor.

One of the first bit-parallel algorithms produced was done by Baeza-Yates and Gonnet[1] in 1992. They developed an algorithm with a time complexity of $O(n\lceil m/w \rceil)$ for the use with exact matching and a $O(n\lceil mlogk/m \rceil)$ for the case of an arbitrary number of mismatches k. Also in 1992, Wu and Manber[18] produced a k mismatch algorithm that ran in a time of $O(nk\lceil m/w\rceil)$. These were developed specifically for applications in text-retrieval and therefore involved small target strings making m small enough compared to w such that the expression between the ceiling braces in the time complexity functions is 1. In this use case, the execution time for the algorithms is $O(kn)$. Wright[17] demonstrated an algorithm with a $O(n\log_2\alpha\lceil m/w\rceil)$ running time. He used 3 bits per character in the bit-parallel encoding. This provided a degree of parallelization of 21 when used with a 64-bit computing system and a relatively small alphabet size α of 8. In 1996 Baeza-Yates and Navarro[2] improved on the algorithm by Wu and Manber[18] to a time complexity of $O(n\lceil km/w\rceil)$. This implementation executed in linear time $O(n)$ for instances when the product of km is $\leq w$. For many implementations, this was the expected use case. By this time, bit-parallel algorithms had become among the fastest methods for approximate string matching.

In 1999, Gene Myers[14] published a bit-parallel algorithm that performed with an even better time complexity of $O(n\lceil m/w\rceil)$. This implementation was proven to have outperformed all its predecessors in nearly all cases. Because of the encoding method used, searches with relatively small target size m compared to the search string n run in linear time $O(n)$. Because of its efficiency, this algorithm was the base bit-parallel implementation chosen for the work in this thesis. Chapter 2 explains in detail the design and implementation used by Myers for this algorithm.

1.2 Bioinformatics

Bioinformatics is a relatively new and fast growing field. It has arguably only been around for about half a century. For this reason, there is no single standard definition for Bioinformatics. The National Cancer Institute defines Bioinformatics as: *"The use of computing tools to manage and analyze genomic and molecular biological data."*. One of the largest affiliations in the field, Bioinformatics.org, defines it as: *"the use of computers to characterize the molecular components of living things"*. A more general definition can describe Bioinformatics as an interdisciplinary area of study that intersects the fields of Biology and Computer Science. It attempts to help collect, process and analyze data generated by the study of living systems. There are several major areas of study in Bioinformatics, including: sequence analysis, comparative genomics, functional genomics, protein expression, protein structure, simulation modeling and gene regulation. In all, Bioinformatics was born out of the need to have efficient computing tools for analyzing the large amounts of data that is generated in these fields of study. Without the parallel advancement of the computing tools used in Biological data analysis, Biological knowledge would likely not be as advanced as it is today.

1.2.1 DNA

Much of the work that concerns Bioinformatics has to do with the study of the processes related to DNA (Deoxyribonucleic Acid). This is the nucleic acid that contains the genetic instructions needed to build the components that make up all living organisms on the planet. Nucleic acids are the macromolecules that dictate the amino acid sequence of proteins, which, in turn, control the basic life processes. They are passed from parent to offspring and also store information that determines the genetic characteristics of cells and organisms. Nucleic acids are made up of nucleotides connected to form long chains. Each consists of three parts. One part is a pentose (5-carbon sugar), which may be either a ribose or deoxyribose. The second part is a phosphate group. The third is a nitrogen base, which is a single or double ringlike structure of carbon, hydrogen, and nitrogen.[4]

Nucleic acids that contain ribose are called ribonucleic acids, or RNA. Those containing deoxyribose form DNA. In DNA, each of the four different nucleotides contains a deoxyribose, a phosphate group, and one of four bases of adenine (A), thymine (T), guanine (G) or cytosine (C). RNA is similar also made up of four bases except that instead of using thymine, it contains uracil (U). DNA polymers can be enormous, containing millions of these nucleotides organized within cells in structures called chromosomes.

DNA was discovered in 1869 by Johann Friedrich Miescher when he isolated a substance he called "nuclein" from the nuclei of white blood cells.[8] For many decades, DNA was largely ignored by biologists because it was not believed to have the ability to code genes. This changed in 1944 when Oswald Avery proved that genes indeed resided within DNA.[8] The next major development that launched the modern era of DNA research occurred in 1953 when James Watson and Francis Crick published a one page paper after discovering the double helical structure of a DNA molecule.[16] This was based partly on the work by Erwin Chargaff who in 1950 discovered that there was a one-to-one ratio of adenine-to-thymine and guanine-to-cytosine content

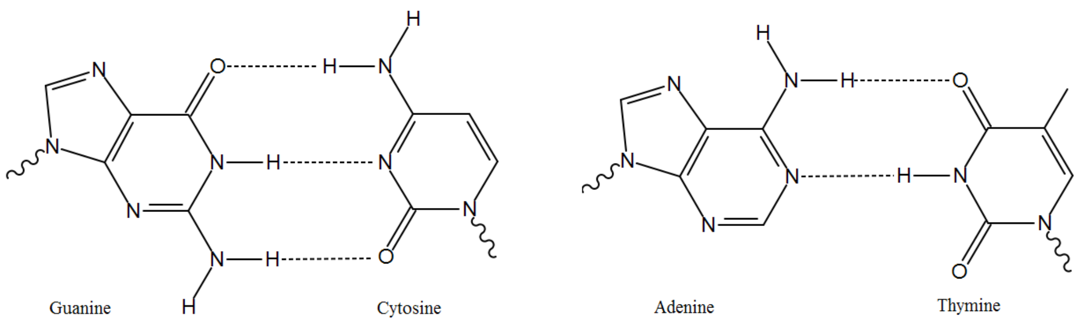


Fig. 1.3: DNA Base Pairing

in DNA, known as the Chargaff rule. Also from the work of Maurice Wilkins and Rosalind Franklin in 1951, when they obtained sharp x-ray images of DNA that suggested DNA was a helical molecule. Watson and Crick arrived at this very simple and elegant double helical structure for DNA after learning that the two strands were held together by hydrogen bonds between specific base pairings: adenine-thymine and guanine-cytosine. Figure 1.3 shows the chemical bonds at work in this base pairing mechanism. This meant that the nucleotide string of one strand defined the nucleotide string of the other. This was the key to understanding DNA replication. Figure 1.4 shows a graphical example of the structure of DNA.

1.2.2 DNA-RNA Connections

There exists a direct connection between DNA and RNA. DNA is the blueprint for the production of hundreds of different kinds of cellular proteins. By controlling protein synthesis, DNA controls the structure and function of cells. Sequences of amino acids form the make up of proteins and define their three dimensional structure. Since DNA directs this sequence of amino acids, it dictates the function of proteins. DNA

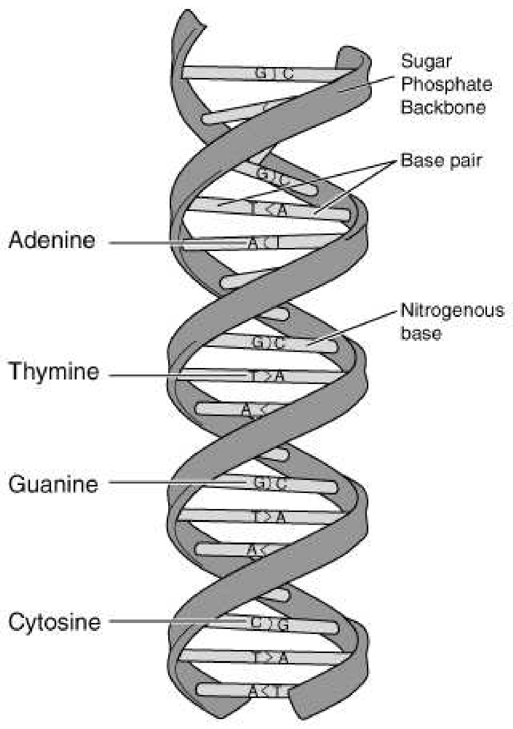


Fig. 1.4: DNA Structure

uses RNA to carry out the actual synthesis of proteins. DNA holds the master set of instructions that is kept secure in the nucleus of the cell. Copies of these instructions are then used to carry the information to structures called ribosomes. The structures that carry this information are called messenger RNA (mRNA). These are synthesized in the nucleus where the master DNA information exists. Two cellular processes, under the direction of DNA, lead to the formation of the primary structure of proteins: *transcription*, or RNA synthesis, and *translation*, or protein synthesis.[4] All proteins that exist are created through this process and flow of information. The sequence is:[8]

$$\text{DNA} \rightarrow \text{transcription} \rightarrow \text{RNA} \rightarrow \text{translation} \rightarrow \text{protein}$$

1.2.3 Genetic Codes

In 1820 Henry Branconnot identified the first amino acid, glycine. Within the next century, all twenty amino acids in existence had been discovered and their chemical structure had been identified. In the early 1900's, Emil Hermann Fischer showed that amino acids were linked together in to linear chains to form proteins. But there was little known at this time as to what processes took place for generating amino acids. The code responsible for the transformation of DNA into proteins was unknown.[8]

Marshall Nirenberg and Heinrich J. Matthaei working at the National Institute of Health in Bethesda, Maryland, made the first major breakthroughs in cracking the genetic code. The key was in mRNA. They conducted a set of experiments using synthetic strands of RNA made up of only uracil. This was added to each of 20 different test tubes containing ribosomes, enzymes and other factors needed for protein synthesis. However, each test tube contained a different radioactive amino acid. In one of the test tubes, the radioactive amino acid was incorporated into polypeptide chains. This is when they realized they had found the genetic code for phenylalanine, three uracil bases in a row (UUU). Biochemists had previously concluded that a triplet of nucleotides, also known as a *codon*, might represent a code that specifies a particular amino acid to be incorporated into a protein. Nirenberg and Matthaei had discovered the first codon. Soon after, all other mRNA codons for amino acids were discovered. With the understanding of codons, a higher degree of prediction for the generation of amino acids was possible. A DNA sequence of n-base pairs will make a protein of a specific $\frac{n}{3}$ amino acids in a specific order. In addition, it is known that there are $4^3 = 64$ different codons. This is many times more than the number of amino acids, which implies that the code for transforming DNA in to protein is degenerate, meaning that different codons can code for the same amino acid.[8] Table 1.3 shows the mRNA genetic code for amino acids. Included in the table are also special types of codons that signal the *start* and *stop* control sequences used during translation.

In cells, sets of DNA bases that code or regulate for particular proteins are called *genes.* These genes are stored in one or more structures called *chromosomes.* Chromosomes can contain thousands of genes in a single long DNA helix. Different organisms have different numbers of chromosomes with different number of genes on each. The complete set of genes in an organism is called its *genome.* Humans have 23 pairs of chromosomes giving a total of 46, and a genome consisting of about three billion DNA bases. Other organisms such as a fruit fly contain only 140 million bases. Ta-

Tab. 1.3: Codon Codes for Amino Acids

Amino Acid	Abbreviation	Codons
Alanine	Ala	GCU, GCC, GCA, GCG
Cysteine	Cys	UGU, UGC
Aspartic acid	Asp	GAU, GAC
Glutamate	Glu	GAA, GAG
Phenylalanine	Phe	UUU, UUC
Glycine	Gly	GGU, GGC, GGA, GGG
Histidine	His	CAU, CAC
Isoleucine	Lle	AUU, AUC, AUA
Lysine	Lys	AAA, AAG
Leucine	Leu	UUA, UUG, CUU, CUC, CUA, CUG
Methionine	Met	AUG
Asparagine	Asn	AAU, AAC
Proline	Pro	CCU, CCC, CCA, CCG
Glutamine	Gln	CAA, CAG
Arginine	Arg	CGU, CGC, CGA, CGG, AGA, AGG
Serine	Ser	UCU, UCC, UCA, UCG, AGU, AGC
Threonine	Thr	ACU, ACC, ACA, ACG
Valine	Val	GUU, GUC, GUA, GUG
Tryptophan	Trp	UGG
Tyrosine	Tyr	UAU, UAC
Start		AUG
Stop		UAG, UGA, UAA

ble 1.4 illustrates a compilation of statistics for the human chromosomes, based on the Sanger Institute's human genome information in the Vertebrate Genome Annotation (VEGA) database. The numbers for genes and bases are estimates based on gene predictions and size of un-sequenced heterochromatin regions.

1.2.4 Coding Errors

The codons contained in mRNA have an interaction with the ribosomes which contain various large molecular structures. The ribosomes are responsible for reading consecutive codons in the mRNA and locating the matching amino acid required for inclusion in the growing polypeptide chain in process. The ribosomes provide much of the physical infrastructure necessary for the production of proteins. To assist in the location of the proper amino acid for a given codon, a particular type of RNA known as transfer RNA (tRNA), is used. Transfer RNA molecules are made up of three base segments called an *anticodon.* Each anticodon is complementary to the codon in mRNA. There are twenty different types of tRNA to match the twenty types of amino acids. Each type of amino acid binds to a particular tRNA. Similar to DNA base pairing, the anticodon on the tRNA sticks to the codon on the RNA, which makes the amino acid available to the ribosome to add to the polypeptide chain. As each amino acid is added, the ribosome shifts one codon to the right, and then repeats the process. This process of turning mRNA into a protein is known as *translation*, because it translates information from the RNA (written in a four-letter alphabet) into the protein (written in a 20-letter alphabet).[8]

This mechanism for producing proteins is not always correct. Proofreading by

Tab. 1.4: Human Chromosomes

Chromosome	Genes	Total bases	Sequenced bases
1	3,148	247,200,000	224,999,719
2	902	242,750,000	237,712,649
3	1,436	199,450,000	194,704,827
4	453	191,260,000	187,297,063
5	609	180,840,000	177,702,766
6	1,585	170,900,000	167,273,992
7	1,824	158,820,000	154,952,424
8	781	146,270,000	142,612,826
9	1,229	140,440,000	120,312,298
10	1,312	135,370,000	131,624,737
11	405	134,450,000	131,130,853
12	1,330	132,290,000	130,303,534
13	623	114,130,000	95,559,980
14	886	106,360,000	88,290,585
15	676	100,340,000	81,341,915
16	898	88,820,000	78,884,754
17	1,367	78,650,000	77,800,220
18	365	76,120,000	74,656,155
19	1,553	63,810,000	55,785,651
20	816	62,440,000	59,505,254
21	446	46,940,000	34,171,998
22	595	49,530,000	34,893,953
X (sex chromosome)	1,093	154,910,000	151,058,754
Y (sex chromosome)	125	57,740,000	22,429,293

enzymes and ribosomes eliminates many errors as proteins are synthesized, but not all are eliminated. Some errors can and do occur in the process. The most common error results from misreading the nucleotide sequence. Initiation determines exactly where translation will begin and how the nucleotide sequence will be grouped into codons. The grouping of bases into codons is called the *reading frame.* If this frame is shifted by one of two nucleotides in any direction, the nucleotide sequence will produce a different sequence of amino acids.[10] For example, AAU GCG GAC UA would specify asparagine-alanine-aspartate. If the reading frame were shifted one nucleotide to the right, the message would read methionine-arginine-threonine. This would not be the intended sequence to be generated.

1.2.5 Mutations

In some cases, errors can occur in the cell nucleus from the DNA itself. Certain segments of DNA are repeated frequently. Some of the repeated segments may become inverted, coding backwards. Some segments mysteriously jump to a new location on the DNA molecule, resulting in incorrect regions along the DNA strand. *Alleles* are alternative forms of a gene that have slightly different base sequences as a result of mutation. A change in the sequence of nucleotide bases in a gene for a particular protein can result in a different sequence of amino acids in the protein. As a result, the three-dimensional structure and, therefore, the function of the protein may be different. The differences in gene function derived from unlike alleles in a population provide the genetic variation on which natural selection can act. An example of alleles is found in the gene that codes for the brown pigment molecules in the iris of the eye.

Brown eyes result from an allele that produces functional pigment. People with blue eyes have alleles of this gene that code for defective pigment molecules. Thus, little pigment is deposited in the iris, and the eyes appear light colored.[4]

Another source of genetic variation are mutations. In animals and many plants, a mutation cannot be passed to the next generation unless it is contained within the chromosomes of a gamete. In plants though, any cell has the potential to produce a new plant, and mutations can be passed on more easily. Mutations occurring in somatic cells, in the body of humans and other animals, may initiate various types of cancers within individuals. This is the reason why *mutagens*, agents that cause mutations, in the environment are of concern to the health of humans and animals.

There exist several types of mutations. In the case of *point mutations*, a single base pair of DNA is improperly paired during replication. Proofreader enzymes in the nucleus check for the correct matching of bases during replication and usually replace any mispaired bases. On occasion, however, a spontaneous mistake escapes detection. The mispairing of bases may be caused by mutagenic chemicals or radiation. Some chemicals resemble bases, are inserted in the forming DNA chain, and then pair with incorrect bases during the next DNA replication. It is a matter of chance on whether or not a single change in base sequence will cause the use of the wrong amino acid sequence of a protein because of the degenerate nature in which codons code for amino acids. In some cases, a single changed nucleotide accidentally forms a stop codon and the resulting polypeptide chain is terminated prematurely.

The most likely types of mutations to disrupt the genetic code are additions and deletions. Adding or removing some bases from the codon sequence, can result in a *frame-shift mutation*, in which the reading frame of the message is altered. Ultraviolet light often causes the formation of covalent bonds between two adjacent thymine nucleotides. The resulting double thymine nucleotide blocks the replication and transcription of DNA. Mutations can also occur in a larger scale in the form of chromosomal alterations. Some of these changes can be attributed to damage caused by ionizing radiation, such as x-rays or gamma-rays. These forms of energy can liberate electrons and create explosively reactive chemicals called *free radicals* that can alter bases in DNA or even tear apart DNA strands. If repair enzymes cannot repair the broken ends, a section of a chromosome may be lost in the next cell division. This type of mutation is known as a *deletion*. Another type is known as *inversion*, where a broken section of DNA sequence is reattached, but in reverse order. *Duplication* occurs when a fragment becomes attached to its homologous chromosome, and the genes contained in the fragment already exist on the homologous chromosome.[4] When broken pieces swap positions on different chromosomes, an event called a *translocation* takes place.

1.3 String Matching In Bioinformatics

String matching plays an essential role as a computational tool in Bioinformatics. This is especially evident in the area of Bioinformatics known as *comparative genomics*. This field of study attempts to exploit both similarities and differences in the DNA, RNA, proteins, and regulatory regions of different organisms to infer how mutations and other aspects of natural selection has acted upon these elements.[5] Comparing genomic sequences is often the key to understanding each of them, which is why recent efforts to sequence many related genomes such as those of humans and chimpanzees provide the best hope for understanding the language of DNA. For example, suppose that we have the genomic sequences of two insects that we suspect are somewhat related in their evolutionary paths; perhaps a fruit fly (*Drosophila melanogaster*) and a malaria mosquito (*Anopheles gamibae*). We would like to know which parts of the fruit fly genomic sequence are dissimilar and what parts are similar to the mosquito genomic sequence. This can help identify linkages in their evolutionary history and can point out the subsets of their genome responsible for the development of the unique physiologies of each species. These can also help to outline how each species has been affected by its environment and therefore evolved to cope with its surroundings.

Comparative genomics can be broken down into several more defined areas of study and into the particular computational problems that can solve each of those. The following subsections introduce some of these challenging problems and how string matching algorithms can be used as tools to solve them.

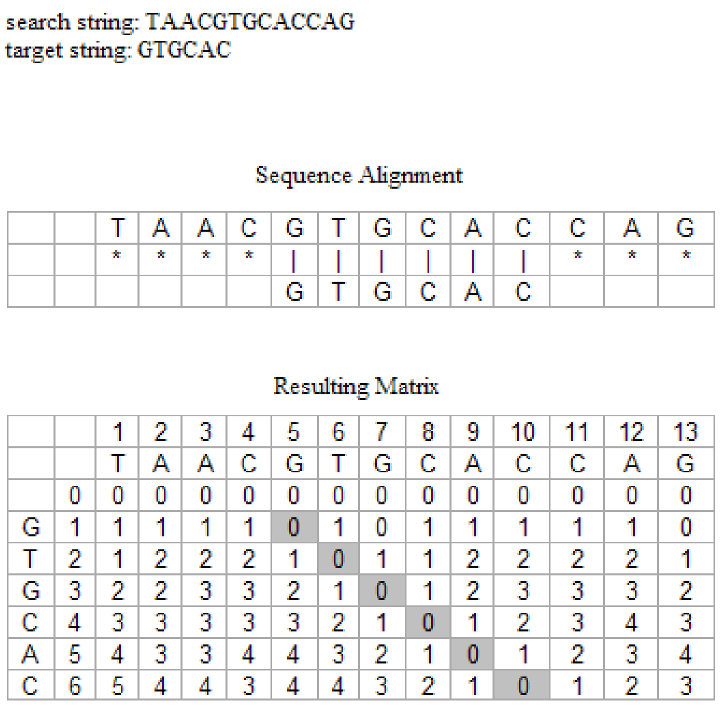
1.3.1 Restriction Mapping

Restriction mapping is the process of obtaining structural information on a piece of DNA by the use of restriction enzymes. Restriction enzymes are enzymes that cut DNA at specific recognition sequences called *sites*. They are believed to have evolved as a bacterial defense against DNA bacteriophage. DNA invading a bacterial cell defended by these enzymes will be digested into small, non-functional pieces. The name *restriction enzyme* comes from the enzyme's function of restricting access to the cell. A bacterium protects its own DNA from these restriction enzymes by having another enzyme present that modifies these sites by adding a methyl group. For example, E.coli makes the restriction enzyme Eco RI and the methylating enzyme Eco RI methylase. The methylase modifies Eco RI sites in the bacteria's own genome to prevent it from being digested.[7]

In 1970 Hamilton Smith discovered that the restriction enzyme HindII cleaves DNA molecules at every occurrence, or site, of the sequence GTGCAC or GTTAAC, breaking a long molecule into a set of restriction fragments.[8] Since this discovery, restriction maps have become a useful tool helping to narrow down the location of certain genetic markers. A restriction map for HindII of a given DNA sequence amounts to finding all occurrences of GTGCAC and GTTAAC. This is clearly a problem that can be solved via a string matching routine by having the restriction site strings as targets and the DNA sequence as the search string. Figure 1.5 shows an example of how the dynamic programming approach of the Levenshtein distance algorithm can be used to find the location of the restriction site GTGCAC in a random DNA sequence.

1.3.2 Motif Finding

The problem of discovering motifs, involves finding approximately repeated patterns in unaligned sequence data. It is important in uncovering transcriptional networks, as short common subsequences in genomic data may correspond to a regulatory protein's binding sites, and in protein function identification, where short blocks of conserved amino acids code for important structural or functional elements.[20] The biological



search string: TAACGTGCACCAG
target string: GTGCAC

Sequence Alignment

		T	A	A	C	G	T	G	C	A	C	C	A	G
		*	*	*	*	|	|	|	|	|	|	*	*	*
						G	T	G	C	A	C			

Resulting Matrix

		1	2	3	4	5	6	7	8	9	10	11	12	13
		T	A	A	C	G	T	G	C	A	C	C	A	G
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	1	1	1	1	1	0	1	0	1	1	1	1	1	0
T	2	1	2	2	2	1	0	1	1	2	2	2	2	1
G	3	2	2	3	3	2	1	0	1	2	3	3	3	2
C	4	3	3	3	3	3	2	1	0	1	2	3	4	3
A	5	4	3	3	4	4	3	2	1	0	1	2	3	4
C	6	5	4	4	3	4	4	3	2	1	0	1	2	3

Fig. 1.5: Example of Levenshtein Distance Algorithm for Finding the Location of a Restriction Site

problems addressed by motif finding are complex and varied, and no single currently existing method can solve them completely. For DNA sequences, motif finding is often applied to sets of sequences from a single genome that have been identified as possessing a common motif, either through DNA micro-array studies, ChIP-chip experiments or protein binding micro-arrays. An orthogonal approach attempts to identify regulatory sites among a set of orthologous genes across genomes of varying phylogenetic distance.[3] For protein sequences, and especially in the case of divergent sequence motifs, it is particularly useful to incorporate amino acid substitution matrices. Often, motif finding methods are either tailor-made to a specific variant of the motif finding problem, or perform very differently when presented with a diverse set of instances. In nearly all cases, some form of string matching algorithm is generally applied as a successful solution.

A practical example can be found in fruit flies whom like humans, are susceptible to infections from bacteria and other pathogens. Although fruit flies do not have as sophisticated an immune system as humans do, they have small set of immunity genes that are usually dormant in the fly genome, but somehow get switched on when the organism gets infected. When these genes are turned on, they produce proteins that destroy the pathogen, usually curing the infection. It turns out that many immunity genes in the frut fly genome have strings that are reminiscent of TCGGGGATTTCC located upstream of the genes' start. These short strings, called NF-kB binding sites, are important examples of *regulatory motifs* that turn on immunity and other genes. Proteins known as *transcription factors* then bind to these motifs, encouraging RNA polymerase to transcribe the downstream genes.[8]

Finding these motifs generally involves searching without prior knowledge of how the motifs sequence is defined. A popular approach to motif finding is based on the assumption that frequent or rare subsequences may correspond to regulatory motifs in DNA. It stands to reason that if a subsequence occurs considerably more frequently then expected, then it is more likely to be some sort of “signal”, and it is crucially important to figure out the biological meaning of that signal.[8]

1.3.3 Longest Common Subsequence

One of the simplest examples of using string matching to solve sequence similarity is the Longest Common Subsequence (LCS) problem. This is important in Bioinformatics because many times it is useful to know what is the largest subset that is common to two sequences. A common subsequence of two strings is one that is a subsequence of both of them. More formally, a common subsequence of strings $s = s_1...s_n$ and $t = t_1...t_n$ is defined as a sequence of positions in s,

$$1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq n$$

and a sequence of positions in t,

$$1 \leq j_1 \leq j_2 \leq \dots \leq j_k \leq m$$

such that the symbols at the corresponding positions in s and t coincide:

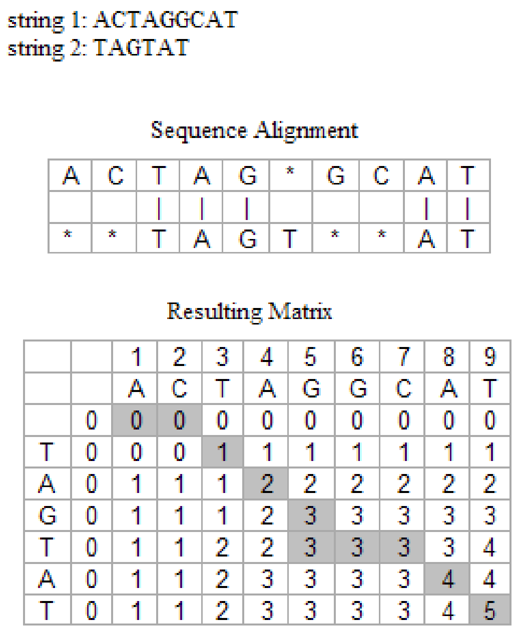
$$s_{i_w} = t_{j_w} \; for \; 1 \leq w \leq k$$

The LCS can be solved by slightly modifying the standard recurrence of the Levenshtein distance algorithm to only allow insertions and deletions and turning it into a maximization problem. In fact, the use of the Levenshtein distance algorithm with different scoring mechanisms for matches, substitutions, additions and deletions can be used to solve a variety of string matching problems. The recurrence to solve the LCS can be formulated as follows:

- Build an $(m+1)$ * $(n+1)$ matrix D, indexed as $D[i][j]$
- Initialize D[*][0] = 0, D[0][*] = 0
- If $p[i] = t[j]$, $D[i][j] = D[i\text{-}1][j\text{-}1] + 1$
- If $p[i] \neq t[j]$, $D[i][j] = \max(\ D[i\text{-}1][j],\ D[i][j\text{-}1])$
- Report the value of $D[m][n]$

Where p = first string, n = size of first string, t = second string, m = size of second string.

Figure 1.6 illustrates the resulting matrix of a sample LCS computation. The last cell with a resulting value of five represents the number of the longest common subsequence between the two strings. This can also be seen in the illustration above the resulting matrix in the sequence alignment, where five characters of each string are matched according to the conditions. The shaded cells show the shortest route that can be taken to obtain the similarity score.



string 1: ACTAGGCAT
string 2: TAGTAT

Sequence Alignment

A	C	T	A	G	*	G	C	A	T
		|	|	|				|	|
*	*	T	A	G	T	*	*	A	T

Resulting Matrix

		1	2	3	4	5	6	7	8	9
		A	C	T	A	G	G	C	A	T
	0	0	0	0	0	0	0	0	0	0
T	0	0	0	1	1	1	1	1	1	1
A	0	1	1	1	2	2	2	2	2	2
G	0	1	1	1	2	3	3	3	3	3
T	0	1	1	2	2	3	3	3	3	4
A	0	1	1	2	3	3	3	3	4	4
T	0	1	1	2	3	3	3	3	4	5

Fig. 1.6: Example of Computing the Longest Common Subsequence Between Two Strings

2. MYERS' BIT-PARALLEL ALGORITHM

One of the most effective techniques developed for performing approximate string matching operations is the use of bit-parallel algorithms. As introduced in chapter 1 section 1.1, bit-parallel algorithms take advantage of the processing of bit-vectors that is done by modern processors. They pack multiple pieces of information into a single word that can be computed atomically. This enables the computation of several units of data at the same time in a single operation and therefore generating a parallel solution. One of the most effective bit-parallel algorithms known to date is the one published by Gene Myers in 1999.[14] The algorithm design demonstrated by Myers, was the one chosen and implemented as the bit-parallel component for the approximate string matching solution presented in this thesis. This chapter describes in depth the design and implementation of the bit-parallel algorithm by Myers. First, section 2.1 describes the ideas behind the design of the bit-parallel approach. The details behind implementing the basic algorithm with a limitation of a maximum target string the size of a machine word is shown in section 2.2. Finally, section 2.3 shows how to extend the basic algorithm to be used without limitation of the size of the target string.

2.1 The Design

The inspiration for the Myers' algorithm comes from using the 4-Russians approach, but using bit-vector computation instead of table lookup. The basic idea behind the design, is to use bit-vectors to encode the list of m arithmetic differences between successive entries in a column of the dynamic programming matrix that is generated during the string matching computation. The operations computed using the bit-vector approach are logical and not arithmetic for a more efficient encoding. In fact, the design uses a single bit to express each unit of data, making the design fully bit-parallel. The time complexity for this approximate string matching algorithm is $O(n\lceil m/w \rceil)$.

The design of the bit-parallel solution assumes the standard description for the approximate string matching problem. We assume to have a query sequence $P = \{p_1 p_2 ... p_m\}$, a search text string $T = \{t_1 t_2 ... t_n\}$, and that we are given a positive threshold $k \geq 0$. We define $\delta(P, T)$ to be the unit cost edit distance between strings P and T. Formally, the approximate string matching problem is to find all positions j in T such that there is a suffix of $T[1...j]$ matching P with a maximum of k differences, that is j such that $\min_g \delta(P,\ T[g...j]) \leq k$.

The standard approach to this problem as we saw in chapter 1 section 1.1, is to compute an $(m+1) \times (n+1)$ dynamic programming (d.p.) matrix $C[0...m, 0...n]$ for which it will be true that $C[i,j] = \min_g \delta(P[1...i,\ T[g...j])$ at the end of the computation. This operation can be computed in $O(mn)$ time and $O(mn)$ space using the recurrence

$$C[i,j] = min \left\{ \begin{array}{c} C[i-1,j-1] + (\ if\ p_i = t_i\ then\ 0\ else\ 1), \\ C[i-1,j]+1, \\ C[i,j-1]+1 \end{array} \right\}$$

subject to the boundary condition that $C[0,\ j] = 0$ for all j. It follows that the solution to the approximate string matching problem is all locations j such that $C[m,\ j] \leq k$.

A basic observation of from this basic implementation as illustrated in chapter 1 section 1.1 is that the computation above can be done in only $O(m)$ space because computing column $C_j = \langle C[i,\ j]\rangle_{i=0}^{m}$ only requires knowing the values of the previous column C_{j-1}. This leads to the important conceptual realization that one may think of a column C_j as a state of an automaton, and the algorithm as advancing from state C_{j-1} to state C_j as it scans symbol t_j of the text. The automaton is started in the state $C_0 = \langle 0,\ 1,\ 2,\ \ldots,\ m\rangle$ and any state whose last entry is $\leq k$ is considered to be a final state.

In 1985, Ukkonen[15] showed that the automaton described here has a finite number of states, which is actually at most 3^m. One can acknowledge this by observing that the dynamic programming matrix C has the property that the difference between adjacent entries in any row or any column is either 1, 0, or -1. A lemma providing a more general version of this was first proven by Masek and Paterson in 1980[13] in the context of the first 4-Russians algorithm for string comparison. Formally, we define the *horizontal* and *vertical deltas* as:

$$\textit{horizontal delta } \Delta h[i,j] \text{ at } (i,\ j) = C[i,\ j] \text{ - } C[i,\ j\text{-}1]$$

$$\textit{vertical delta } \Delta v[i,\ j] \text{ at } (i,\ j) = C[i,\ j] \text{ - } C[i\text{-}1,\ j]$$

for all $(i,\ j)\ \epsilon\ [1,\ m]$ x $[1,\ n]$. We then have:

Lemma 1. $\forall(i, j)(\Delta v[i, j], \Delta h[i, j]) \in \{-1, 0, 1\}$

Proof. Originally described by Ukkonen[15]. Since $C[i, j]$ is always an integer, it suffices to show that $C[i, j] - 1 \leq C[i-1, j-1] \leq C[i, j]$. The minimization step in the recurrence directly implies that $C[i, j]$ cannot be larger than $C[i-1, j-1] + 1$ and hence, $C[i, j] - 1 \leq C[i-1, j-1]$. This is trivially true in the base case $C[0, 0]$. We proceed by induction on $i + j$. Assume first that the minimizing path to $C[i, j]$ comes from $C[i-1, j-1]$. Then the recurrence implies that $C[i, j] = C[i-1, j-1]$ or $C[i, j] = C[i-1, j-1] + 1$. Hence $C[i, j] \geq C[i-1, j-1]$, as required. We then assume that the minimizing path to $C[i, j]$ comes from $C[i-1, j]$ along with the similar symmetric case where the path comes from $C[i, j-1]$. Then again by the recurrence, $C[i, j] = C[i-1, j]+1$. By induction hypothesis $C[i-1, j] \geq C[i-2, j-1]$. Hence $C[i, j] \geq C[i-2, j-1] + 1$. Finally, since $C[i-1, j-1 \leq C[i-2, j-1] + 1$ by the recurrence, this implies that $C[i, j] \geq C[i-1, j-1]$ as required.

It follows that, to know a particular state C_j, it suffices to know the *relocatable* column $\Delta v_j = \langle \Delta v[i,\ j] \rangle_{i=1}^{m}$ because $C[0,\ j] = 0$ for all j. One can see that the automaton can have at most 3^m states as there are 3 choices for each vertical delta.

The problem of computing the cell values in the dynamic programming matrix C can thus be replaced with that of computing the *relocatable* dynamic programming matrix Δv. One potential difficulty in this method is that determining if Δv_j is final requires $O(m)$ time as one must determine if $\Sigma_i \Delta v_j[i] = C[m,\ j] \leq k$. While this does not effect the asymptotics of most algorithmic variations on the basic d.p. formulations, it is crucial to algorithms such as this one, which compute a block of

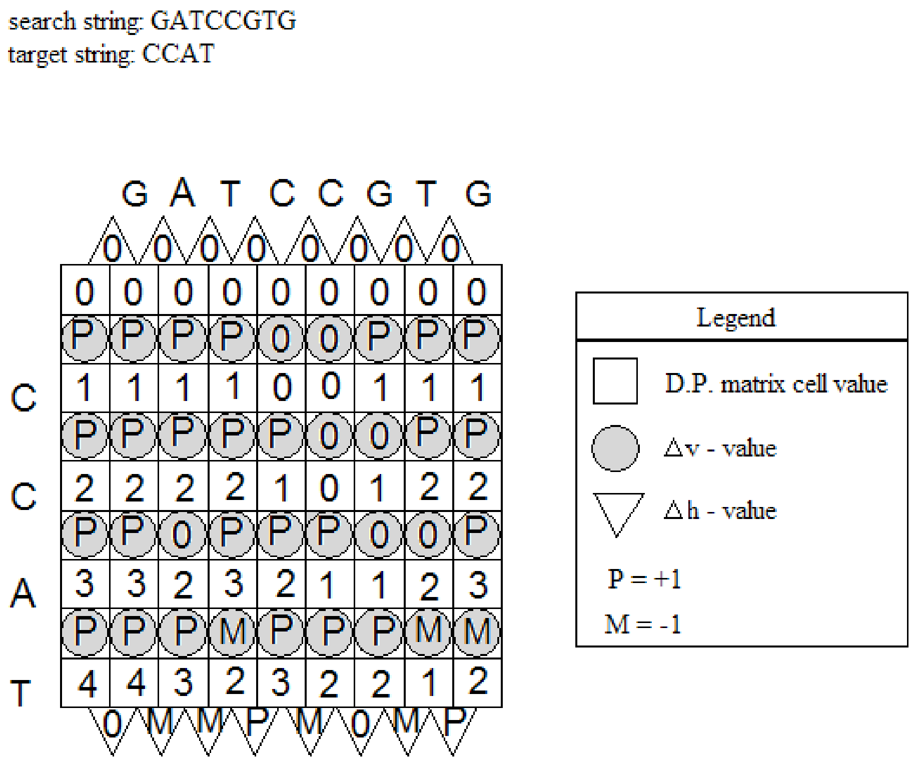


Fig. 2.1: Illustration of a Bit-Parallel Dynamic Programming Matrix

vertical deltas in $O(1)$ time and thus cannot afford to compute the sum over these deltas without affecting both their symptotic and practical efficiency. Fortunately, we can simultaneously maintain the value of $Score_j = C[m, j]$ as one computes the Δv_j values using the fact that $Score_0 = m$ and $Score_j = Score_{j-1} + \Delta h[m, j]$. The *horizontal delta* in the last row of the matrix is required, but it will be shown later, this delta at the end of the block of vertical delta's is a natural by-product of the block's computation. Figure 2.1 shows the basic dynamic dynamic programming matrix and the formulation of the delta values.

2.2 Developing the Basic Algorithm

The essential idea behind the algorithm is to compute successive Δv_j values in $O(1)$ time by using bit-vector operations. This is where the algorithm makes great performance gains by performing the operations in parallel.

2.2.1 Assumptions

It is assumed for the rest of the description of this algorithm that the size of a machine word is w and that $m \leq w$. On most current machines, the word size is usually 32 or 64. It is also assumed that parallel bit operations as in *or, and* and *not*, as well as simple arithmetic operations of addition and subtractions take the underlying computing system constant time to complete on such words.

2.2.2 Representation

One of the basic challenges is to define the bit-vector representation for Δv_j. This is accomplished by using two bit-vectors Pv_j and Mv_j, whose bits are set according to whether the corresponding value in Δv_j is +1 or -1, respectively. Formally,

$$Pv_j(i) \equiv (\Delta v[i, j] = +1)$$
$$Mv_j(i) \equiv (\Delta v[i, j] = \text{-}1)$$

where the notation *W(i)* denotes the i^{th} bit of the word W, and where i is assumed to be in the range $[1, w]$. It must be noted that the i^{th} bits of vectors Pv and Mv must not be set simultaneously, and that we do not need a vector to encode the positions i that are zero, as we know they occur when $(\neg(Pv_j(i) \vee Mv_j(i)))$ is true.

2.2.3 Cell Structure

It is necessary to develop an understanding of how to compute the deltas in one column from those in the previous column. To start, consider an individual *cell* of the d.p. matrix consisting of the square (*i-1, j-1*), (*i-1, j*), (*i, j-1*), and (*i, j*). There are two horizontal and vertical deltas: $\Delta v[i,j], \Delta v[i,j-1], \Delta h[i,j]$ and $\Delta h[i-1,j]$ that are associated with the sides of this cell as illustrated in Figure 2.2(a). Further, let *Eq*[*i, j*] be a bit quantity which is 1 if $p_i = t_j$ and 0 otherwise. Using the definition of the deltas and the basic recurrence for *C-values* we arrive at the following equation for $\Delta v[i,j]$ in terms of $Eq[i,j], \Delta v[i,j-1]$ and $\Delta h[i-1,j]$:

$$\Delta v[i,j] = C[i,j] - C[i-1,j]$$

$$\Delta v[i,j] = min \left\{ \begin{array}{c} C[i-1,j-1] + (\ if\ p_i = t_i\ then\ 0\ else\ 1), \\ C[i-1,j]+1, \\ C[i,j-1]+1 \end{array} \right\} - C[i-1,j]$$

$$\Delta v[i,j] = min \left\{ \begin{array}{c} C[i-1,j-1] + (1 - Eq[i,j]), \\ C[i-1,j-1] + \Delta v[i,j-1] + 1, \\ C[i-1,j-1] + \Delta h[i-1,j] + 1 \end{array} \right\} - (C[i-1,j-1] + \Delta h[i-1,j])$$

$$\Delta v[i,j] = min \left\{ \begin{array}{c} -Eq[i,j], \\ \Delta v[i,j-1], \\ \Delta h[i-1,j] \end{array} \right\} + (1 - \Delta h[i-1,j])$$

Similarly:

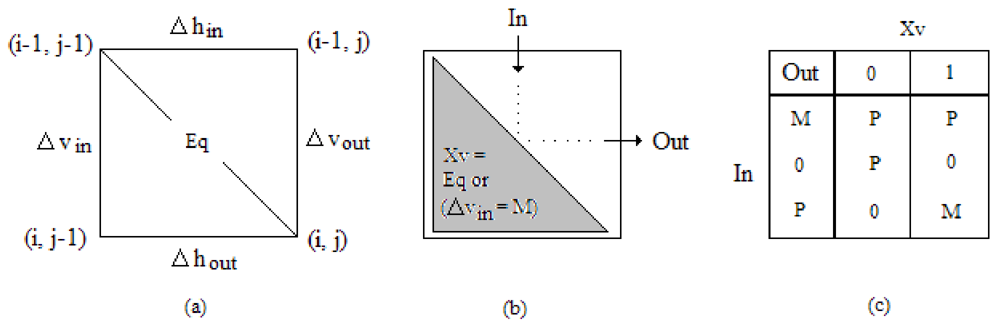


Fig. 2.2: Bit-Vector Cell Structure and Input/Output Function

$$\Delta h[i,j] = min \left\{ \begin{array}{c} -Eq[i,j], \\ \Delta v[i,j-1], \\ \Delta h[i-1,j] \end{array} \right\} + (1 - \Delta v[i,j-1])$$

We can then view the inputs to a cell as:

$$\Delta v_{in} = \Delta v[i,j-1], \Delta h_{in} = \Delta h[i-1,j], Eq = Eq[i,j]$$

and the outputs as:

$$\Delta v_{out} = \Delta v[i,j], \Delta h_{out} = \Delta h[i,j]$$

2.2.4 *Cell Logic*

There exist three choices for each of Δv_{in} and Δh_{in} and two possible values for *Eq*. This implies that there is a finite number of inputs possible for a given cell, 18. From this, evolved the key idea that one could compute the numeric values in a column with Boolean logic.

It is conceptually easier to think of Δv_{out} as a function of Δh_{in} modulated by an auxiliary Boolean value *Xv* capturing the effect of both Δv_{in} and *Eq* and Δv_{out}. This is illustrated in Figure 2.2(b). By using a brute force enumeration of the 18 possible inputs, it is possible to verify the correctness of the table presented in Figure 2.2(c) which describes $\Delta\mathrm{v}_{out}$ as a function of $\Delta\mathrm{h}_{in}$ and *Xv*. In the table, the value -1 is denoted by M and +1 by P, in order to emphasize the logical, as opposed to the numerical, relationship between the input and output. Let Px_{io} and Mx_{io} be the bit values encoding $\Delta\mathrm{x}_{io}$, that is, $Px_{io} \equiv (\Delta\mathrm{x}_{io} = +1)$ and $Mx_{io} \equiv (\Delta\mathrm{x}_{io} = \text{-}1)$. From the table, the following logical formulas capturing this function can be verified:

$$\begin{aligned} Xv &= Eq \vee Mv_{in} \\ Pv_{out} &= Mh_{in} \vee \neg(Xv \vee Ph_{in}) \\ Mv_{out} &= Ph_{in} \wedge Xv \end{aligned} \tag{2.1}$$

The following symmetric formulas for computing the bits of the encoding of $\Delta\mathrm{h}_{out}$ are given by the relationship between Δh_{out} and Δv_{in} modulated by
$Xh \equiv (Eq \vee (\Delta h_{in} = -1))$:

$$\begin{aligned} Xh &= Eq \vee Mh_{in} \\ Ph_{out} &= Mv_{in} \vee \neg(Xh \vee Pv_{in}) \\ Mh_{out} &= Pv_{in} \wedge Xh \end{aligned} \tag{2.2}$$

2.2.5 Preprocessing the Alphabet

The Boolean value of $Eq[i, j]$ for each cell $(i,\ j)$ is necessary during the evaluation process. To represent this using bit-vectors, we need an integer Eq_j for which
$Eq_j(i) \equiv (p_i = s)$. The computation of these integers during the scan would require $O(m)$ time which is not what we expect from this solution. Fortunately, we can

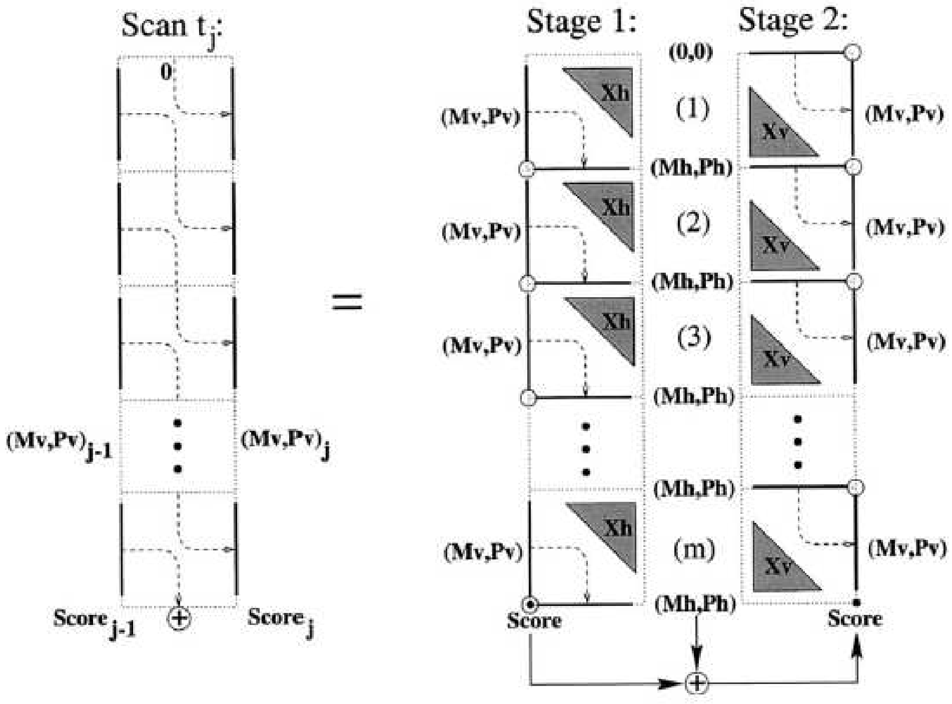


Fig. 2.3: Bit-Parallel Scanning Stages

perform a preprocessing step before the scan begins and compute a table of the vectors that result for each possible text character. Formally, if α is the alphabet which makes up all characters in P and T, then we build an array $Peq[\alpha]$ for which:

$$Peq[s](i) \equiv (p_i = s) \tag{2.3}$$

Assuming that α is finite, he table construction can be achieved in $O(|\alpha| + m)$ time and it occupies $O(|\alpha|)$ space.

2.2.6 The Scanning Step

The central inductive step is to compute $Score_j$ and the bit-vector pair (Pv_j, Mv_j) encoding $\Delta \mathrm{v}_j$, given the same information at column *j-1* and symbol t_j. The concept of the automata is kept and this step is referred to as *scanning* t_j. The basis of the induction is as follows:

$$\begin{aligned} Pv_0(i) &= 1 \\ Mv_0(i) &= 0 \\ Score_0 &= m \end{aligned} \tag{2.4}$$

Meaning that at the start of the scan, the value of *Score* is *m*, the bit-vector *Mv* is all 0's and the *Pv* bit-vector is all 1's.

The difficulty presented by the induction step is that given the vertical delta on its left side, the only applicable formulas, namely (2.2), give the horizontal delta at the bottom of the cell, whereas the goal is to have the vertical delta on its right side. This can be accomplished in two stages as shown in Figure 2.3. First, the vertical delta values in column *j-1* are used to compute the horizontal delta values at the bottom of their respective cells, using formula (2.2). Second, the horizontal delta values are used in the cell below to compute the vertical deltas in column *j*, using formula (2.1).

The *Score* in the last row is updated between the two stages using the last horizontal delta now available from the first stage, and then the horizontal deltas are all *shifted* by one, pushing out the last horizontal delta and introducing a 0-delta for the first row. Each stage serves as a pivot, where the pivot of the first stage is at the lower left of each cell, and the pivot of the second stage is at the upper right. The delta values swing in the arc depicted and produce results modulated by the relevant X values. The computation of *Xh* and *Xv* is presented in subsection 2.2.7.

The logical formulas (2.1) and (2.2) for a cell and the illustration in Figure 2.3, lead directly to the formulas below for accomplishing a scanning step. It must be noted that the horizontal deltas of the first stage are recorded in a pair of bit-vectors, (Ph_j, Mh_j), that encodes horizontal deltas exactly as (Pv_j, Mv_j) encodes vertical deltas as $Ph_j(i) \equiv (\Delta\text{h}[i,\ j] = +1)$ and $Mh_j(i) \equiv (\Delta\text{h}[i,\ j] = \text{-}1)$.

$$\begin{aligned} Ph_j(i) &= Mv_{j-1}(i) \vee \neg(Xh_j(i) \vee Pv_{j-1}(i)) \\ Mh_j(i) &= Pv_{j-1}(i) \wedge Xh_j(i) \end{aligned} \tag{Stage 1}$$

$$Score = Score_{j-1} + (1\ if\ Ph_j(m)) - (1\ if\ Mh_j(m)) \tag{2.5}$$

$$\begin{aligned} Ph_j(0) &= Mh_j(0) = 0^2 \\ Pv_j(i) &= Mh_j(i-1) \vee \neg(Xv_j(i) \vee Ph_j(i-1)) \\ Mv_j(i) &= Ph_j(i-1) \wedge Xv_j(i) \end{aligned}$$

(Stage 2)

It is important to note that the formulas above specify the computation of bits in bit-vectors, all of whose bits can be computed in parallel with the appropriate machine operations.

2.2.7 Computing the X-Values

To complete the induction presented in the last subsection, we must show how to compute the bits of the bit-vectors Xv_j and Xh_j. From the logical formulas (2.1) and (2.2) we have:

$$\begin{aligned} Xv_j(i) &= Peq[t_j](i) \vee Mv_{j-1}(i) \\ Xh_j(i) &= Peq[t_j](i) \vee Mh_j(i-1) \end{aligned} \tag{2.6}$$

Where we make use of the pre-computed table *Peq* to lookup the necessary *Eq* bits. The computation of Xv_j at the beginning of the scan is simple, since Mv_{j-1} is an input to the step. The computation of Xh_j is more problematic because it requires the value of Mh_j which in turn requires the value of Xh_j. This cyclic dependency must be unwound. Lemma 2 gives this formulation of Xh_j which depends only on the values of Pv_{j-1} and $Peq[t_j]$.

Lemma 2. $(\exists k \leq i)(\forall x \in [k, i-1])(Xh_j(i) = Peq[t_j](k) \wedge Pv_{j-1}(x))$

Proof. From formulas (2.1) and (2.1) for all k, $Mh_j(k)$ is true iff $Pv_{j-1}(k)$ and $Xh_j(k)$ are true. By combining this with equation (2.6), it follows that
$Mh_j(k) \equiv ((Pv_{j-1}(k) \wedge Peq[t_j](k)) \vee ((Pv_{j-1}(k) \wedge Mh_j(k-1))$. By repeatedly applying this, we obtain the desired statement by induction:

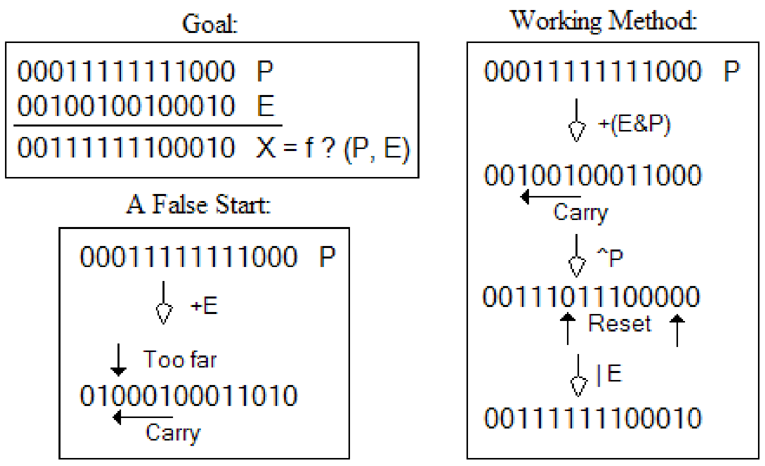


Fig. 2.4: Computing Xv

$$
\begin{aligned}
Xh_j(i) &= Peq[t_j](i) \vee Mh_j(i-1) \\
&= Peq[t_j](i) \vee (Pv_{j-1}(i-1) \wedge Mh_j(i-2)) \\
&\quad \vee (Pv_{j-1}(i-1) \wedge Mh_j(i-2)) \\
&= Peq[t_j](i) \vee Pv_{j-1}(i-1) \wedge Peq[t_j](i-1)) \\
&\quad \vee (Pvj-1(i-1) \wedge Pvj-1(i-2) \wedge Peq[t_j](i-2)) \\
&\quad \vee (Pvj-1(i-1) \wedge Pvj-1(i-2) \wedge Mh_j(i-3)) \\
&= \ldots \\
&= (\exists k \leq i)(\forall x \in [k, i-1])(Peq[t_j](k) \wedge Pv_{j-1}(x)) \qquad (as\ Mh_j(0) = 0)
\end{aligned}
$$

The last remaining obstacle is to determine how to compute the bit-vector Xh in a constant number of word operations. Lemma 2 states that the i^{th} bit of Xh is set whenever there is a preceding Eq bit, say the k^{th} and a run of set Pv bits covering the interval $[k, i-1]$. It is useful to think of the Eq bit as being "propagated" along the run of set Pv bits, setting positions in the Xh vector as it does so. Addition of integers has a similar effect on the underlying bit encoding. This is illustrated in Figure 2.4. First, consider the effect of adding P and E together, where P has the value of Pv_{j-1} and E that of $Peq[t_j]$. Each bit in E initiates a carry-propagation chain down a run of set P-bits that turns these bits to 0's except where an E-bit is also set. In the figure, this possibility is labeled "A False Start" because we observe that the carry propagation can proceed beyond the end of a run of set P-bits because of set E-bits. Therefore, one must first turn off all E-bits that are not covered by a run of set P-bits as E & P, and then add this to P. It is possible to then capture all the bits in P that have been toggled during the carry propagation by taking the exclusive *or* of the result with P. Finally, it is possible to *or* in the E-bits to capture those that were either not covered by a run of set P-bits, or that were not the initiators of a carry propagation chain. From this, the following formula is derived and verified by Lemma 3:

$$Xh_j(i) \quad = \quad (((Peq[t_j](i) \wedge Pv_{j-1}) + Pv_{j-1}) \oplus Pv_{j-1}) \vee Peq[t_j]) \qquad (2.7)$$

Lemma 3.

If $X = (((E \wedge P) + P) \oplus P) \vee E,$ *then* $(\exists k \leq i)(\forall x \in [k, i-1])(X(i) = E(k) \wedge P(x))$

Proof. Figure 2.5 illustrates the mechanics of an addition automaton. A transition of the form *a, b/c* is taken when the corresponding bits of the operands are a and b, and bit c results. A 1 is output when in the *Carry* state iff the bits of the operands are equal. The opposite is output if the automaton is in the *No Carry* state. Furthermore, one is in the *Carry* state when processing bit i iff there is a previous bit position k, for which the its of both operands are set and where at least one of the operands has

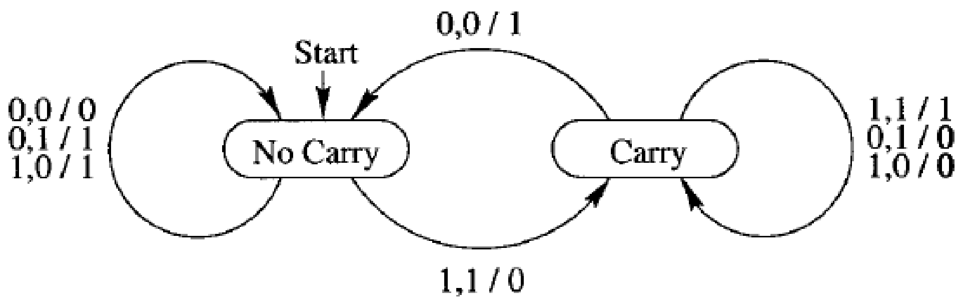


Fig. 2.5: The Addition Automaton

bits set in all positions between k and i. This leads to the formal logical description of the effect of addition:

$(\exists k < i)(\forall x \in [k, i-1])((Q+P)(i) = Q(k) \wedge P(k) \wedge (Q(x) \vee P(x))) \equiv (Q(i) \equiv P(i))$

Replacing Q by $E \wedge P$ in this expression and then applying some simple logical inferences leads to the conclusion that, if $y = (E \wedge P) + P$, then:

$(\exists k < i)(\forall x \in [k, i-1])(Y(i) = E(k) \wedge P(x)) \equiv (E(i) \vee \neg(P(i)))$

The next step is to use the interfaces that $((A \equiv B) \oplus (P)\ iff\ (A \equiv (B \oplus P))$ and that $((E \vee \neg(P) \oplus P)\ iff\ \neg(E \wedge P)$, to conclude that, if $Y = ((E \wedge P) + P) \oplus P$, then:

$(\exists k < i)(\forall x \in [k, i-1])(Y(i) = E(k) \wedge P(x)) \equiv (\neg(E(i)) \wedge P(i))$

The last step requires the interface $((A \equiv B) \vee E) \wedge ((\neg B) \rightarrow E)$ is equivalent to $(A \vee E)$. This is true for all cases except when $(A = 1, B = 0, E = 0)$. Meaning that, if $Y = (((E \wedge P) + P) \oplus P) \vee E$, then it follows that:

$(\exists k < i)(\forall x \in [k, i-1])(Y(i) = E(k) \wedge P(x) \vee E(i))$,

which is only a slight restatement of the conclusion of the lemma.

Tab. 2.1: The Basic Bit-Parallel Algorithm in Pseudocode

Line	Pseudocode	Formula
1.	Precompute $Peq[\alpha]$	2.3
2.	$Pv = 1^m$	
3.	$Mv = 0$	2.4
4.	$Score = m$	
5.	for $j = 1, 2, ...n$ do	
6.	{	
7.	$Eq = Peq[t_j]$	
8.	$Xv = Eq \vee M$	2.6
9.	$Xh = (((Eq \wedge Pv) + Pv) \oplus Pv) \vee Eq$	2.7
10.	$Ph = Mv \vee \neg(Xh \vee Pv)$	
11.	$Mh = Pv \wedge Xh$	2.2
12.	if $Ph \wedge 10^{m-1}$ then	
13.	$Score$ += 1	
14.	else if $Mh \wedge 10^{m-1}$ then	2.5
15.	$Score$ -= 1	
16.	$Ph \ll= 1$	
17.	$Mh \ll= 1$	
18.	$Pv = Mh \vee \neg(Xv \vee Ph)$	2.1
19.	$Mv = Ph \wedge Xv$	
20.	if $Score \leq k$ then	
21.	"Match found at ", j	
22.	}	

2.2.8 The Complete Algorithm

At this stage, it is possible to place all the pieces together to complete the algorithm. Table 2.1 shows a complete specification in pseudocode. The table Peq is constructed prior to the scan as specified by formula (2.3). The bit-vectors, Pv and Mv and the integer $Score$ are maintained during the scan and at the completion of scanning the j^{th} character contain the values of Pv_j, Mv_j and $Score_j$, respectively. These are computed according to the formula (2.4) to correctly initiate the scan. To scan the symbol t_j, the algorithm uses five intermediate bit-vectors Eq, Xv, Xh, Ph, and Mv in the interior of the scan loop. First, Xh and Xv are computed to have the values of Xh_j and Xv_j according to formulas (2.6) and (2.7) using the variable Eq to factor the common subexpression $Peq[t_j]$. Then Ph and Mh are computed to hold the horizontal deltas for the j^{th} column with formula (2.2), $Score$ is updated to the value of $Score_j$ using formula (2.5), and Pv and Mv are updated to hold the vertical deltas in column j. Finally, the value of $Score$ is checked to see if there is a match. The complexity of the algorithm is $O(m\alpha + n)$ where α is the size of the alphabet α. Only 17 bit operations are performed per character scanned.

The last thing to consider is the case where m is unrestricted. This can be accomplished by modeling an m-bit bit-vector with $\lceil m/w \rceil$ words. An operation on such bit-vector takes $O(m/w)$ time. It follows that the basic algorithm described here runs in $O(m\alpha + nm/w)$ time and $O(\alpha m/w)$ space.

2.3 General Target Length Extension

It is possible to extend the basic algorithm to a more general case where we can solve for target strings with a size larger than w, the word size. It is necessary to understand how to encapsulate the result of the basic algorithm into modules or *blocks* that can be pieced together to solve larger problems. Similar to the input/output function of a cell with its four deltas at its borders, it is possible to more generally think of the computation of a $u \times v$ rectangular sub-array or *block* of cells as resulting in the output of deltas along its lower and right boundary, given deltas along its upper and left boundary as input.

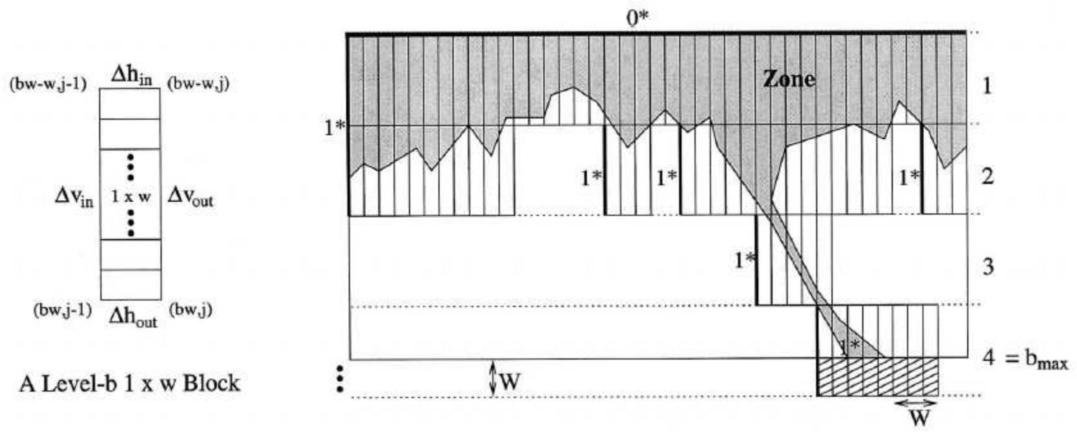


Fig. 2.6: Block-Based Dynamic Programming

We can think of the basic algorithm as affecting the $O(1)$ computation of $1 \times m$ blocks under the special circumstances that the horizontal input delta is always 0. More generally, we can use the result to effect the computation of $1 \times w$ blocks where the horizontal input delta may also be -1 or +1. The left diagram in Figure 2.6 illustrates such a block and terms it a *level b* block since it extends from row $(b-1)w$ to row bw. If we limit out attention to only blocks on $O(m/w)$ levels, we are still able to cover any desired region of a d.p. matrix, and only $O(\alpha m/w)$ *Eq*-vectors need be pre-computed.

The diagram to the right of Figure 2.6 shows a d.p. matrix and a hypothetical zone that might be computed by an algorithm that can compute a partial region or zone of a dynamic programming matrix. It illustrates how it is possible to take any such underlying computation and perform it in fewer steps by computing the region w cells at a time. This kind of tiling involves at most $b_{max} = \lceil m/w \rceil$ levels. The following are additional aspects that must be considered:

1. The computation can proceed in a column sweep so that only b_{max} vertical delta vectors need to be maintained at any one time, such that we can think of the *current* vertical delta at level b.

2. The blocks at the boundaries of the matrix have deltas of either 0 or 1 depending on the underlying computation. In Figure 2.6 0-deltas are depicted on the upper boundary and 1-deltas on the left boundary of the matrix.

3. For blocks that have no predecessor at the same level in the previous column can usually assume 1-deltas for their vertical inputs, as this conservatively models values greater than those in the zone.

4. In the last level, blocks may extend beyond the last row by $W = w - m \pmod{w}$ cells. The simplest method to handle this case is to pad the length m sequence with W extra wild-card symbols. The value of the interior horizontal delta in a row m, then appears at the output of the level-b_{max} block W columns later. This delay in output requires that one also pad the length n sequence with W wild-card symbols, and that one extend a tiling W columns beyond the end of the zone when in this last level, as shown in Figure 2.6.

3. MULTIPROCESSOR PARALLELISM

We have seen that bit-parallelism can improve the time complexity of the base algorithm used for approximate string matching. The parallel aspect of the bit-parallel algorithm is to take advantage of the ability to perform the computation of multiple data units in parallel on a single processor. This chapter presents a design feature used in addition to bit-parallelism; the use of multiple processors in parallel. The idea for this approach is to further minimize the total time required to complete a string matching operation by having multiple processors work simultaneously to solve a single problem while using the bit-parallel algorithm. Section 3.1 introduces the basic methods and challenges involved in order to successfully distribute the problem to multiple processors. The implementation for the working solution is presented in section 3.2. Finally, section 3.3 shows empirical results and analysis of that data.

3.1 Distributing the Problem

After analysis into the nature of the dynamic programming approach used in the bit-parallel algorithm by Myers[14], it is possible to see that a multiprocessor design should have great potential for efficiency. But before we can explore how to distribute the problem of approximate string matching among several processors, it is necessary to note some of the challenges involved in the process. There are several factors that we must consider when designing this multiprocessor solution.

The main issue we must figure out is what work the different processors will perform in parallel. We know that the computation of each cell in the dynamic programming matrix depends only on the values of the current and previous columns. This low level of data dependency between computed columns in the matrix means that the construction of the columns can be logically partitioned into segments to be computed by different processors simultaneously. The simplest approach is to partition the search string into p subsets of equal size, where p is the number of processors. This means that for a search string of size n, we would divide it into p subsets each of size n/p. Ideally, each processor would independently work on its subset of search string data and when complete simply report the locations in the subset that match the target string.

This approach is efficient and requires no communication overhead between the different processors, but there is one adjustment we must make in order to maintain correctness. The problem is that even though the computation of each column in the dynamic programming matrix only depends on the values of itself and the previous column, the values contained in that previous column were derived from the column previous to it. In other words, the *Score* that is maintained at the end of each column in the matrix that is relevant for knowing if a match has occurred, depends on more than just the current and previous column. Figure 3.1 illustrates this scenario. It shows a comparison of the computation of the dynamic programming matrix using single and dual processors for search string t="GTTTACGTTGAGTGTGCG" of size $n = 18$, target string p="ATTG" of size $m = 4$ and a maximum mismatch k of 1.

search string: GTTTACGTTGAGTGTGCG
target string: ATTG
k: 1

Sequence Alignment

G	T	T	T	A	C	G	T	T	G	A	G	T	G	T	G	C	G
						x	|	|	|	|	x	|	|				
						*	T	T	G	A	*	T	G				

Single Processor - Resulting Matrix

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		G	T	T	T	A	C	G	T	T	G	A	G	T	G	T	G	C	G
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1
T	2	2	1	1	1	1	1	2	1	1	2	1	1	1	2	1	2	2	2
T	3	3	2	1	1	2	2	2	2	1	2	2	2	1	2	2	2	3	3
G	4	3	3	2	2	2	3	2	3	2	1	2	2	2	1	2	2	3	3

Dual Processor - Resulting Matrix

P1

		1	2	3	4	5	6	7	8	9
		G	T	T	T	A	C	G	T	T
	0	0	0	0	0	0	0	0	0	0
A	1	1	1	1	1	0	1	1	1	1
T	2	2	1	1	1	1	1	2	1	1
T	3	3	2	1	1	2	2	2	2	1
G	4	3	3	2	2	2	3	2	3	2

P2

		10	11	12	13	14	15	16	17	18
		G	A	G	T	G	T	G	C	G
	0	0	0	0	0	0	0	0	0	0
A	1	1	0	1	1	1	1	1	1	1
T	2	2	1	1	1	2	1	2	2	2
T	3	3	2	2	1	2	2	2	3	3
G	4	3	3	2	2	1	2	2	3	3

Fig. 3.1: Illustration of Multiprocessor Left Boundary Problem

The sequence alignment shows that there are two instances of the target string in the search string with a mismatch of 1 each. The resulting matrix for the single processor case shows that the location for each match is at 10 and 14, respectively. The bottom diagram shows two resulting matrices after distributing the search to two processors without taking into account the issue of the derived value of *Score*. We can see that the matrix for the second processor $P2$, fails to locate the match on location 10. This is because the starting values in the left-most column are not equivalent to those in the right-most column of the matrix produced by $P1$. Therefore, by using the incorrect default values in the *left boundary* of the matrix in $P2$, successive columns are generated with incorrect results.

The question then becomes: how many previous columns relative to the current column must we consider at any given time to maintain a correct *Score* value? The answer is $m - 1$, where m is the size of the target string. We can prove this by analyzing the basic recurrence used in the approximate string matching algorithm.

$$C[i,j] = \min\{C[i\text{-}1,\ j\text{-}1] + (\text{if } p_i = t_j \text{ then } 0 \text{ else } 1),\ C[i\text{-}1,\ j] + 1,\ C[i,\ j\text{-}1] + 1\}$$

Since the concern with the *left boundary* issue when segmenting the search string is that we may miss a match in the current column due to not carrying over the values of cells for previous matching characters, we must specifically pay close attention to the part of the recurrence that accounts for a match. When a match occurs in the current cell, it simply takes on the value of the cell nearest in the previous row and column:

$$\text{If } p_i = t_j, \text{ then } C[i,j] = C[i-1, j-1]$$

Due to the nature of the direct diagonal movement for matching values in cells, such a value in a cell at the origin of the dynamic matrix can affect a maximum of m columns. Since we don't need to account for the current column as it is the one we are currently computing, the overlap needed to fix the *left boundary* problem is only the previous $m - 1$ columns. Figure 3.2 shows an example at work. The diagram at the bottom illustrates that by appending $m - 1$ characters to the left of the search string subset for $P2$, we are able to maintain the proper $Score$ values at the bottom row of the resulting matrix and therefore not missing the match at location 8. The *left boundary* overlap only needs to take place for processor subsets that do not start at the beginning of the search string. Therefore, $P1$ in the example does not require any overlap. In all, it is necessary to overlap a total of $(m-1) \times (p-1)$ characters, where p is the number of processors being used. It is important to note that this design becomes impractical for use-cases where the target string is similar in size to the search string. Therefore, it is assumed that the use-cases targeted by this solution are those where $m \ll n$. In practice, this can be used in a variety of real-world applications in Bioinformatics where the target strings tend to be much smaller than the search strings.

3.2 The Implementation

The system architecture for this bit-parallel/multiprocessor solution is based on the use of MPI (Message Passing Interface), executing on an underlying network of heterogeneous workstations. MPI is a software layer that allows for the management of parallel processes in a distributed memory system. This means that all sharing of data between processors must take place through the use of message passing. Through this

search string: GTTTACGTTG ($n = 10$)
target string: ACGT ($m = 4$)
k = 0

Sequence Alignment

		G	T	T	T	A	C	G	T	T	G
		*	*	*	*	|	|	|	|	*	*
						A	C	G	T		

Single Processor - Resulting Matrix

		1	2	3	4	5	6	7	8	9	10
		G	T	T	T	A	C	G	T	T	G
	0	0	0	0	0	0	0	0	0	0	0
A	1	1	1	1	1	0	1	1	1	1	1
C	2	2	2	2	2	1	0	1	2	2	2
G	3	2	3	3	3	2	1	0	1	2	2
T	4	3	2	3	3	3	2	1	0	1	2

Dual Processor - Resulting Matrix
(without left boundary overlap)

P1

		1	2	3	4	5
		G	T	T	T	A
	0	0	0	0	0	0
A	1	1	1	1	1	0
C	2	2	2	2	2	1
G	3	2	3	3	3	2
T	4	3	2	3	3	3

P2

		6	7	8	9	10
		C	G	T	T	G
	0	0	0	0	0	0
A	1	1	1	1	1	1
C	2	1	2	2	2	2
G	3	2	1	2	3	2
T	4	3	2	1	2	3

Dual Processor - Resulting Matrix
(with (m-1) left boundary overlap)

P1

		1	2	3	4	5
		G	T	T	T	A
	0	0	0	0	0	0
A	1	1	1	1	1	0
C	2	2	2	2	2	1
G	3	2	3	3	3	2
T	4	3	2	3	3	3

P2

					6	7	8	9	10
		T	T	A	C	G	T	T	G
	0	0	0	0	0	0	0	0	0
A	1	1	1	0	1	1	1	1	1
C	2	2	2	1	0	1	2	2	2
G	3	3	3	2	1	0	1	2	2
T	4	3	3	3	2	1	0	1	2

m - 1 Overlap

Fig. 3.2: Overlapping $m - 1$ Characters on the Left Boundary

MPI interface, the same copy of a program can be executed in a predefined number of processors simultaneously. At run-time, each processor is uniquely identified in sequence as $\{p_0, p_1, ..., p_{s-1}\}$, where s is the number of processors being used. Using the unique identifier, processes can direct communication or use broadcasting methods for the purpose of sharing data, synchronization and other functions.

In this implementation, there is no explicit communication between processes. The data used by the program on each processor, is assumed to be available on a shared file on disk or redundant in a local disk to each processor. Each processor runs an exact copy of the approximate string matching program. At run-time, each copy of the program on the individual processors dynamically calculates the subset of the search string it is responsible for working on. It then streams the necessary data to local memory from an available file. Each processor completes its own alphabet preprocessing and then performs the bit-parallel algorithm on the particular subset of search string it is responsible for and maintains the locations of matches for the given target string and threshold.

Using this design which applies virtually no unnecessary overhead, the bit-parallel algorithm performed in a multiprocessor approach is shown to produce nearly perfect parallel efficiency according to empirical results presented in the next section.

3.3 Empirical Results

This section presents empirical results gathered during a variety of experiments conducted. The bit-parallel/multiprocessor program was tested on the *raven* cluster which is hosted in the computer science department at the California State University San Bernardino. The cluster is made up of 13 Compaq Proliant DL-360 G2 machines. Each containing two Pentium III processors running at 1.4 GHz with 256MB of SDRAM. The cluster is connected via switched 1000GB Ethernet. Each machine has access to both local and shared SCSI disks. The operating system on each is Linux OS with Kernel version 2.4.20-8smp. The particular version of MPI used is mpich-1.2.5.2. All program code was written in the ANSI C language and compiled with the *mpicc* compiler. The complete program is made up of a single source file named `bp_mp.c` and is fully available in Appendix A.

During testing, the *raven* cluster was fully dedicated to the given experiments. The experiments tested for parallel performance on a number of variables including, different number of processors p, different search string lengths n, different target string lengths m and different mismatch thresholds k. The two basic measurements used to analyze the results of tests conducted are *speedup* and $efficiency$. Speedup is defined as the measure of how much faster an algorithm can execute in parallel compared to the same algorithm running in sequential form. The formula for speedup is defined as:

$$S_p = \frac{t_1}{t_p}$$

Where p is the number of processors, t_1 is the sequential execution time and t_p is the parallel execution time. We can see that the optimal speedup would be equal to the number of processors to which the algorithm is being distributed to. Efficiency is the measure of real processor utilization during parallel execution. It is defined as:

$$E_p = \frac{S_p}{p}$$

The range of possible values are between [0,1], with 1 being optimal efficiency.

During each experimental run, time measurements were gathered for individual runs with a given number of processors. Each data element gathered for time execution on each processor represents the average value of 10 total runs. From this data, the average and worse case executions were computed. Given a set of resulting average execution times for p processors

$$T = \{t_1, t_2, ..., t_p\},$$

the average execution time between processors is computed as:

$$T_{ave} = \tfrac{(t_1+t_2+...+t_p)}{p}$$

and the worse case time as:

$$T_{max} = \max(t_1, t_2, ..., t_p)$$

The tests were arranged such that a single variable was modified per test in order to properly direct the cause of any observed changes in performance. The search and target strings for all tests were made up of the alphabet $\alpha = \{A, C, G, T\}$. The calculated results are presented here in a graphical form. Each figure contains two graphs. The graph on the left shows a comparison of the worse case, average, and ideal execution times using a base 10 logarithmic scale for the vertical axes and linear scale for the horizontal axes. The graph on the right shows both the worse case, average and ideal speedup along with the efficiency computed using the worse case speedup values. This graph uses linear scaling in both the horizontal and vertical axes. Prior to each result graph, a table demonstrates the mean from the collected test data set. This is the value used for the average case in each plotted point on the time graph, in seconds. The table also shows error analysis via the standard deviation for the collected data set of 10 runs.

Figures 3.3, 3.4, 3.5, 3.6, 3.7 illustrate results using non-changing target string of size m=16, mismatch threshold k=0, and processor range between [1-7] with the variable being the size of the search string data. The results show nearly optimal results for the first four cases. The exception is the last experiment with the 100MB search string size. It this case, parallel usage with more than three processors results in an increasing drop in overall speedup for an ultimate drop of about 22% from optimal when using seven processors. The reason for this decrease in parallel performance is the usage of a single shared data source for all processors. All processors access the same storage hard disk to stream their respective subsets of the search string. Disk access is the main bottleneck of the implemented solution.

One way to diminish the disk access bottleneck is to pre-distribute the search string data to local disks for each processor. This can be easily done on a system that uses a static number of processor nodes. Each processor then will have exclusive access to its part of the search string data, eliminating collisions due to simultaneous access. The amount of data used for left boundary patching can be estimated during pre-distribution and adjusted at run-time if needed at a low cost. This was left as a future performance improvement to the solution and not implemented here.

The group of figures 3.8, 3.9, 3.10 and 3.11 show resulting graphs for a variation in the mismatch threshold k, while maintaining a static search string with n=25MB, target string with m=16 and processor range of [1-7]. This set of experiments reveals that the results are maintained at optimal parallel performance in all cases, showing that a variation in the mismatch threshold k does not affect the parallel performance of the solution.

The final group of graphs in figures 3.12 and 3.13 present a variation in the target string size m, while maintaining static search string size n=25MB, mismatch threshold k=0 and a processor range of [1-7]. Here as in the previous experiment group, the variation in target string shows that the parallel efficiency of the solution is not affected and results are maintained at a nearly optimal level. As noted in section 3.1, unless the target string becomes similar in size to the to the search string, the effectiveness of the parallel distribution should remain highly efficient.

Tab. 3.1: Result Data for n=1MB, m=16, k=0, p=[1-7]

Processor Count	1	2	3	4	5	6	7
Mean Time	0.058767	0.029579	0.019840	0.015023	0.012019	0.010042	0.008804
Standard Dev.	1.4640e-4	6.7528e-5	3.4554e-5	1.4840e-5	1.5815e-5	6.3073e-6	2.0820e-5

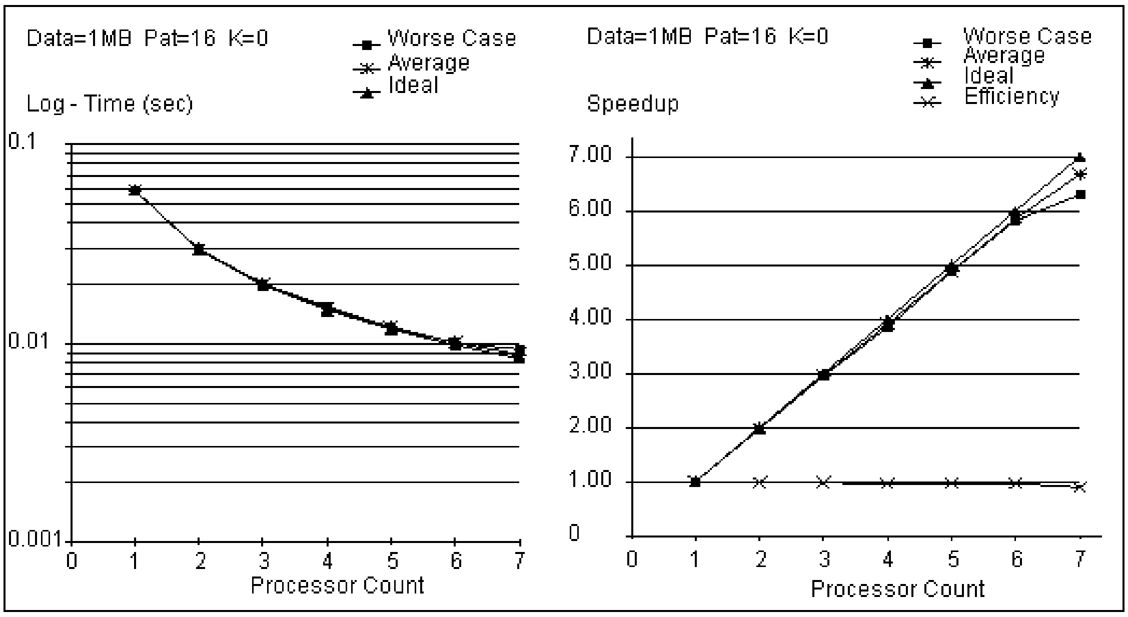


Fig. 3.3: Result Graphs for n=1MB, m=16, k=0, p=[1-7]

Tab. 3.2: Result Data for n=10MB, m=16, k=0, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	0.58222	0.292271	0.195112	0.156071	0.123302	0.100374	0.086539
Standard Dev.	5.6863e-4	3.4502e-4	2.0661e-4	2.1278e-4	1.5246e-4	9.6198e-5	7.4856e-5

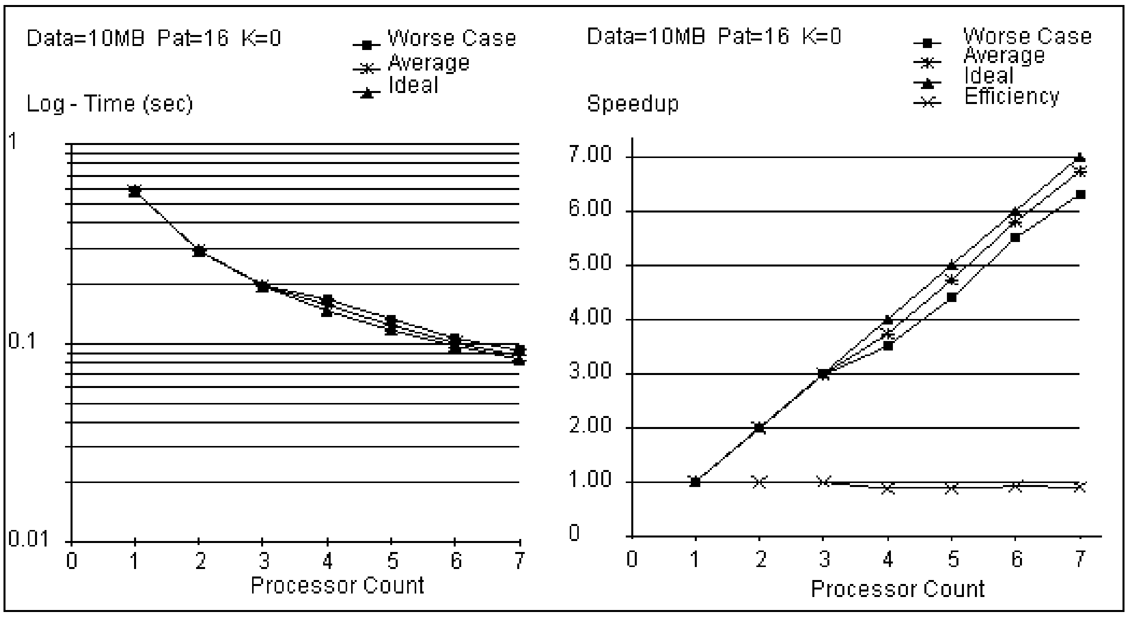


Fig. 3.4: Result Graphs for n=10MB, m=16, k=0, p=[1-7]

Tab. 3.3: Result Data for n=25MB, m=16, k=0, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	1.45406	0.729919	0.508467	0.403623	0.302807	0.247854	0.20933
Standard Dev.	1.9193e-3	1.1386e-3	7.3016e-4	3.7637e-3	6.0077e-4	4.0204e-4	1.8306e-4

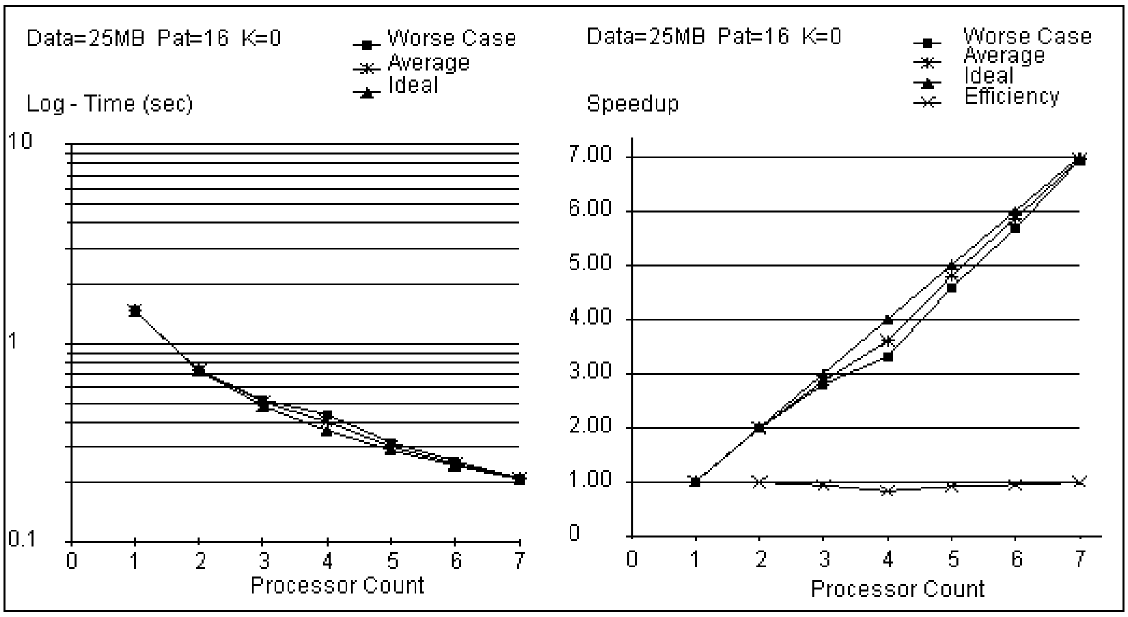


Fig. 3.5: Result Graphs for n=25MB, m=16, k=0, p=[1-7]

Tab. 3.4: Result Data for n=50MB, m=16, k=0, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	2.90805	1.4578	1.0645	0.832698	0.650938	0.525132	0.428855
Standard Dev.	4.5025e-3	1.7785e-3	1.5858e-3	1.4891e-3	6.0121e-4	7.7184e-4	61112e-4

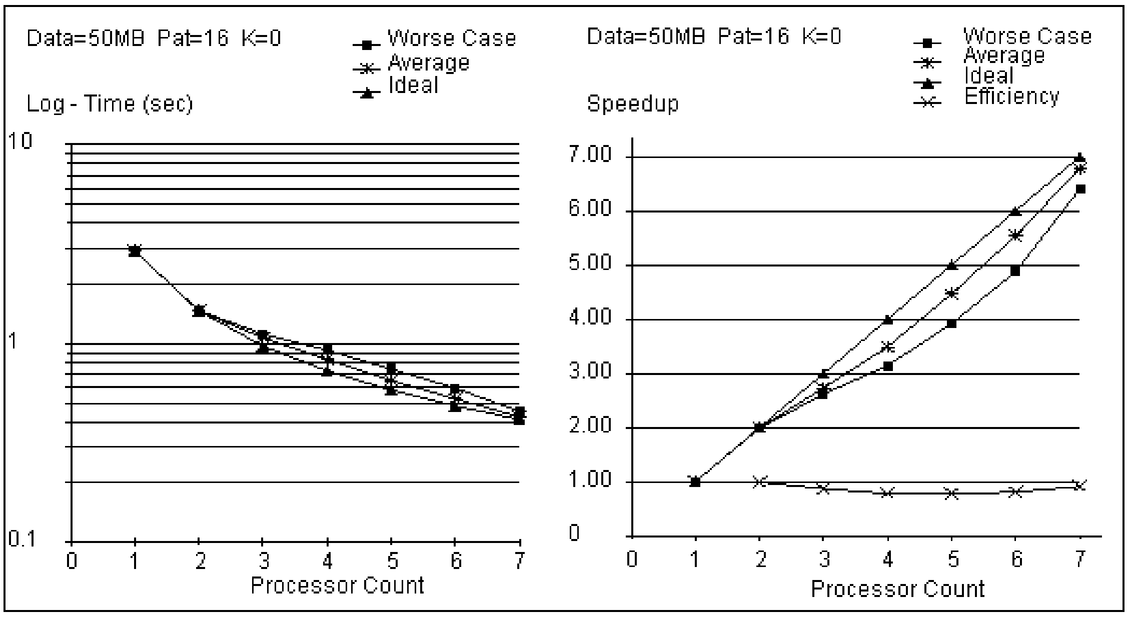


Fig. 3.6: Result Graphs for n=50MB, m=16, k=0, p=[1-7]

Tab. 3.5: Result Data for n=100MB, m=16, k=0, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	5.818791	2.919976	2.135254	1.639896	1.299515	1.063902	0.904460
Standard Dev.	6.1613e-3	4.7189e-3	4.9188e-3	1.3303e-3	2.7741e-3	3.1183e-3	1.6954e-3

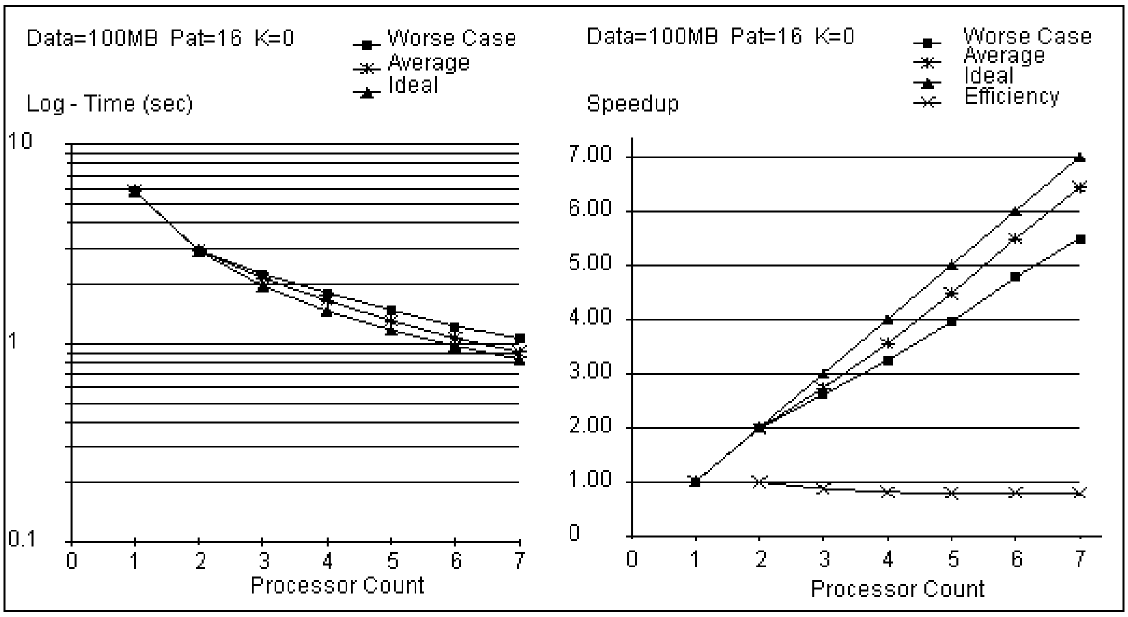


Fig. 3.7: Result Graphs for n=100MB, m=16, k=0, p=[1-7]

Tab. 3.6: Result Data for n=25MB, m=16, k=2, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	1.47633	0.730161	0.521347	0.407431	0.308085	0.249336	0.209404
Standard Dev.	1.8011e-3	1.1405e-3	4.1629e-4	5.6869e-4	5.0279e-4	2.9167e-4	2.8607e-4

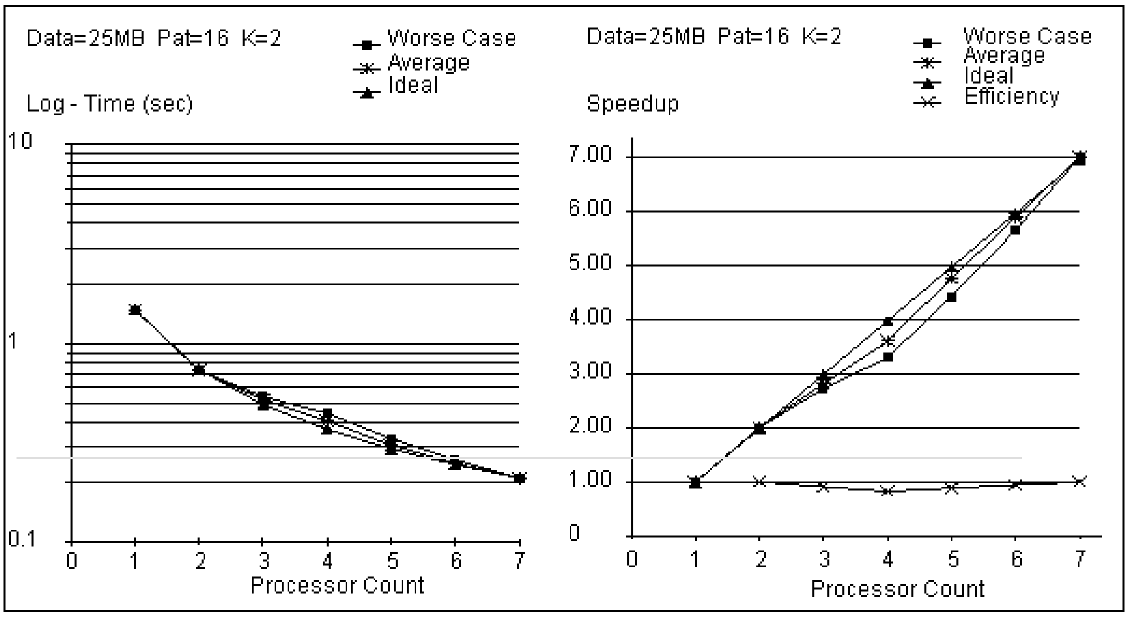


Fig. 3.8: Result Graphs for n=25MB, m=16, k=2, p=[1-7]

Tab. 3.7: Result Data for n=25MB, m=16, k=4, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	1.62783	0.81608	0.569348	0.424531	0.334487	0.277791	0.234083
Standard Dev.	2.6988e-3	8.7157e-4	4.9505e-4	6.3212e-4	6.4631e-4	3.7752e-4	4.0871e-4

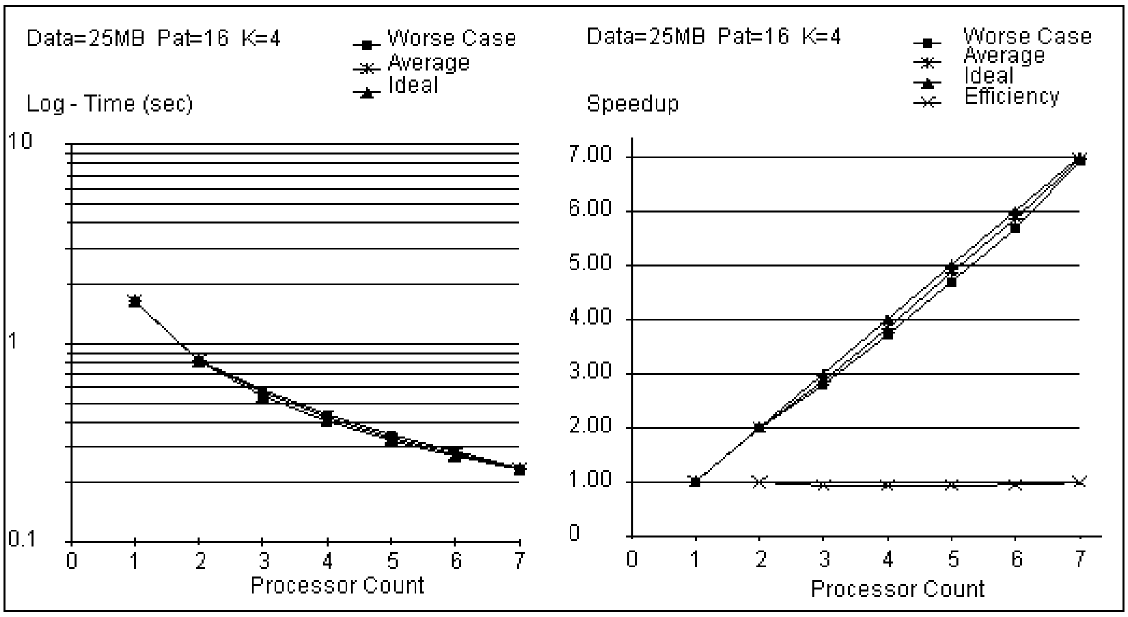


Fig. 3.9: Result Graphs for n=25MB, m=16, k=4, p=[1-7]

Tab. 3.8: Result Data for n=25MB, m=16, k=8, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	2.20491	1.1059	0.757305	0.570285	0.448742	0.373546	0.316612
Standard Dev.	4.1673e-3	1.3454e-3	6.0432e-4	8.2862e-4	6.1433e-4	5.9175e-4	3.3842e-4

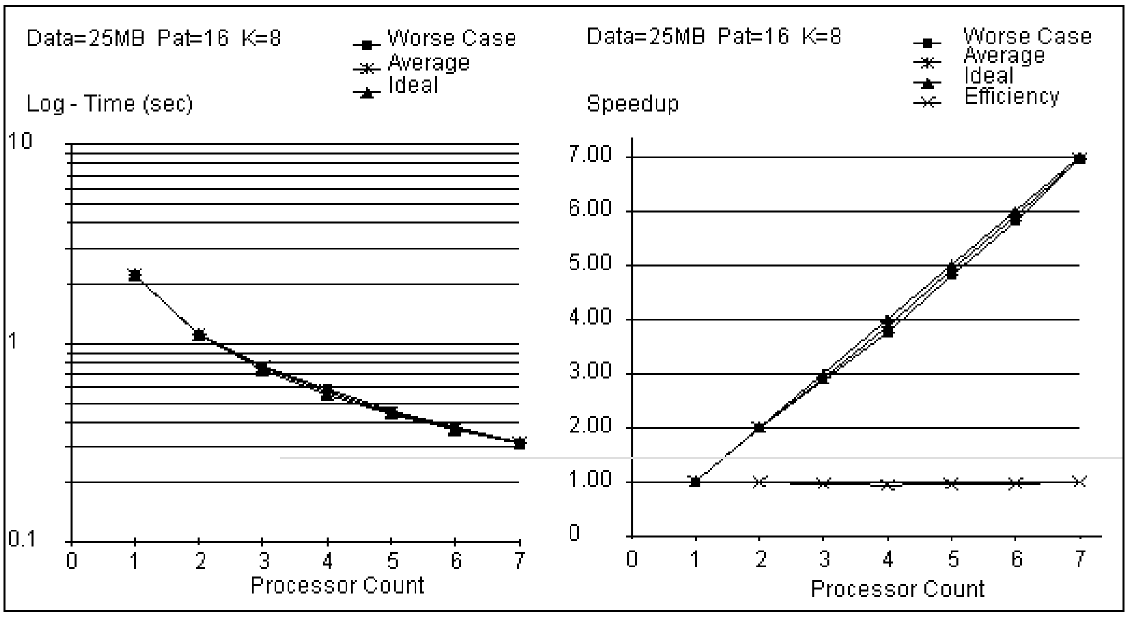


Fig. 3.10: Result Graphs for n=25MB, m=16, k=8, p=[1-7]

Tab. 3.9: Result Data for n=25MB, m=16, k=12, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	2.02305	1.0249	0.697125	0.534951	0.411318	0.340875	0.290851
Standard Dev.	3.0123e-3	1.8018e-3	1.0240e-3	8.8742e-4	4.0568e-4	3.1923e-4	2.6973e-4

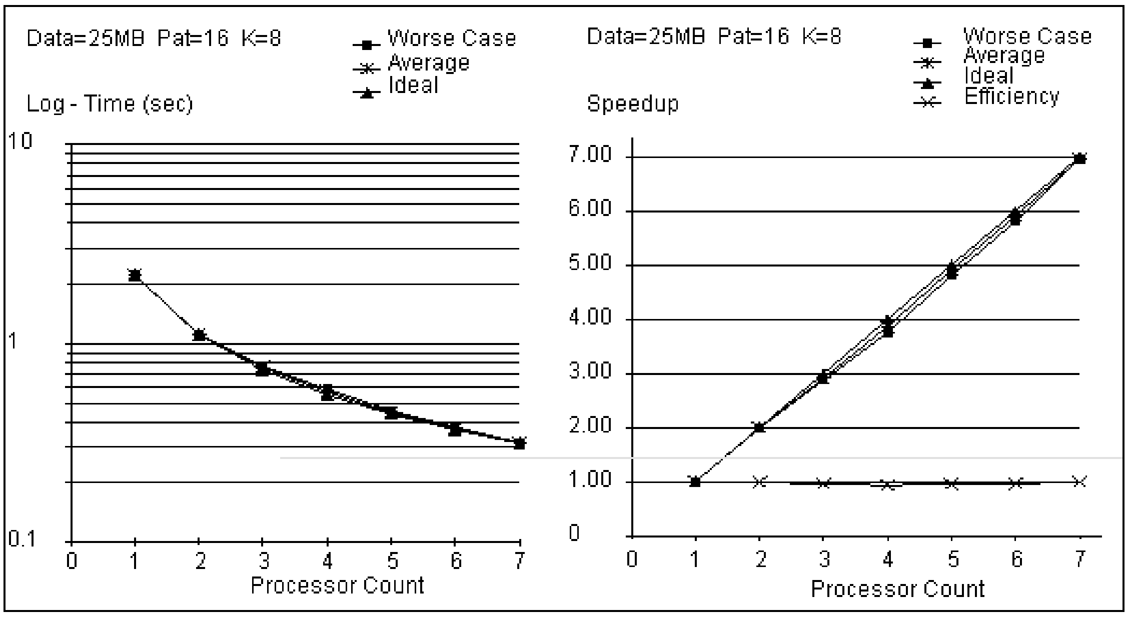


Fig. 3.11: Result Graphs for n=25MB, m=16, k=12, p=[1-7]

Tab. 3.10: Result Data for n=25MB, m=64, k=0, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	1.44943	0.727571	0.485294	0.38491	0.297208	0.244951	0.210891
Standard Dev.	2.1726e-3	1.0186e-3	6.4107e-4	3.8009e-4	4.9277e-4	3.1795e-4	3.2941e-4

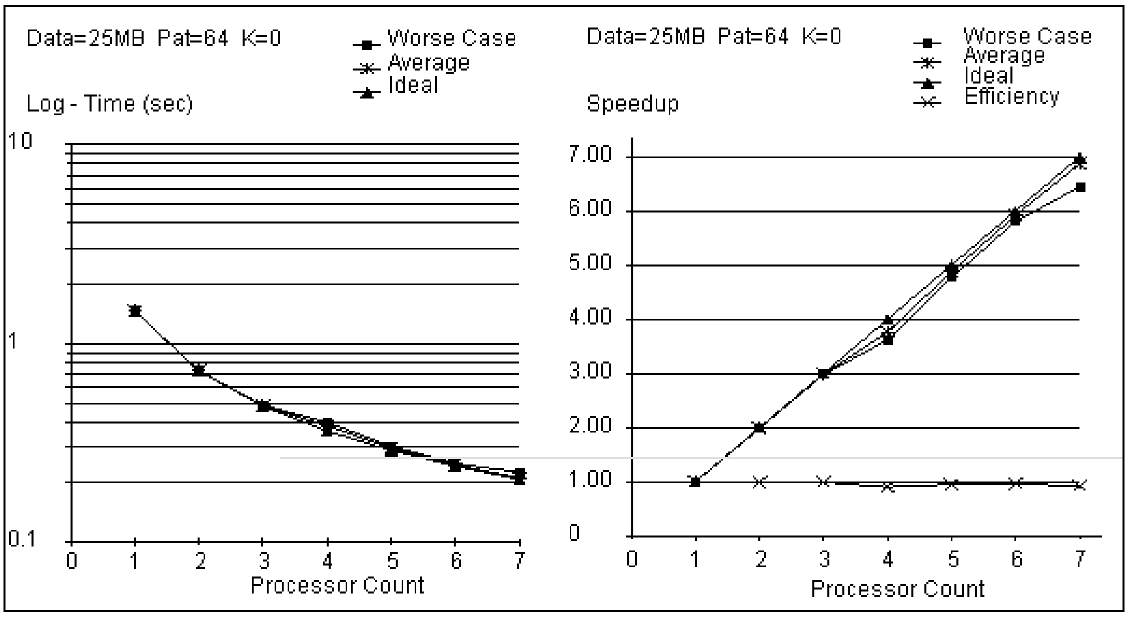


Fig. 3.12: Result Graphs for n=25MB, m=64, k=0, p=[1-7]

Tab. 3.11: Result Data for n=25MB, m=1K, k=0, p=[1-7]

Processors	1	2	3	4	5	6	7
Mean Time	1.46109	0.733322	0.49005	0.386702	0.299655	0.248396	0.211557
Standard Dev.	2.6578e-3	1.3791e-3	6.3991e-4	5.7579e-4	4.6806e-4	3.6315e-4	2.8247e-4

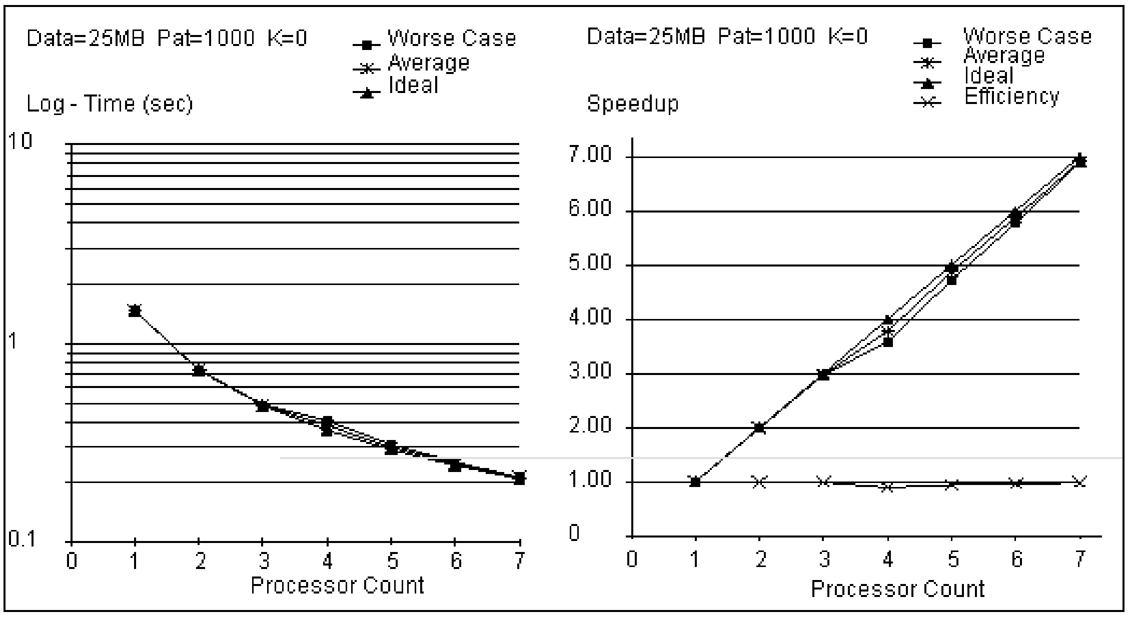


Fig. 3.13: Result Graphs for n=25MB, m=1K, k=0, p=[1-7]

4. CONCLUSION

There is an increasing need for effective, low-latency approximate string matching algorithms for use in a variety of real world applications. This is especially true for the field of Bioinformatics. This ever-growing field relies on the ability of computing systems to serve as efficient tools in the management and analysis of genomic and molecular biological data. This thesis shows that an approximate string matching solution combining both a bit-parallel and multiprocessor design serves as a viable answer. Empirical results prove that for most practical search cases where the size of a target string is relatively small compared to the search string, nearly optimal speedup is achieved.

APPENDIX A

SOURCE CODE

```
/*————————————————————————
bp_mp.c

Author: Elias Chibli, elchibli@csci.csusb.edu

This source file implements a bit-parallel-multiprocessor
algorithm for approximate string matching. The bit-parallel
algorithm was originally designed by Gene Myers and is described in
his 1999 paper titled: "A Fast Bit-Vector Algorithm for Approximate
String Matching Based on Dynamic Programming". The original
algorithm has been modified here to be used in a multiprocessor
parallel environment using MPI.
————————————————————————*/

#include <stdio.h>
#include <stdlib.h>
#include <sys/file.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <sys/time.h>
#include "mpi.h"

//————————————————————————-
//Global variables and definitions

#define CONTROL_PROCESS 0
#define CHAR  0
#define WORD long
//The Alphabet: all possible character values in the 7-bit set
#define SIGMA 128
//Default read block size
#define BUF_MAX  2048

typedef struct { char type, *value; } element;

typedef struct
{
  unsigned WORD P;
  unsigned WORD M;
  int V;
} Scell;

static int       patlen;
static element *patvec;

static int W;
static unsigned WORD All = -1;
static unsigned WORD Ebit;

static unsigned WORD *TRAN[SIGMA];
static unsigned WORD Pc[SIGMA];
static int seg, rem;

//——————————————————————-
//setup_search()
//Performs the basic pre-processing step necessary for fast
//character matching at run time
//
void setup_search()
{
  register unsigned WORD *b, bvc, one;
  register int a; //alphabet loop counter
  register int p; //pattern loop counter
  register int i, k;

  //8 bits in a byte * the number of bytes per word in this machine
  W = sizeof(unsigned WORD)*8;

  //How many WORD segments are needed to store the pattern. The +1 is needed
  //because if the pattern is less than W, the result from the division is 0
  seg = ( ( patlen - 1 )/W ) + 1;

  //This is how many bits are not needed since we only care about those
  //for the pattern
  rem = seg*W - patlen;

  //Allocate (WORD size: 4) * (Alphabet size: 128)
  //* (the segments required to store the target pattern) + 1
  b = (unsigned WORD *) malloc(sizeof(unsigned WORD)*(SIGMA*seg+1));

  //loop over each possible character value (7-bits)
  for (a = 0; a < SIGMA; a++)
  {
    TRAN[a] = b;

    //loop over the pattern. Each loop go around looks
    //at 32 characters, hence the (p+=W)
    for (p = 0; p < patlen; p += W)
        {
      bvc = 0;
      one = 1;
      k = p+W;

      if (patlen < k)
      {
```

```
      k = patlen;
    }

    for (i = p; i < k; i++)
        {
      if (patvec[i].type == CHAR)
            {
        if (a == *(patvec[i].value))
        {
          bvc |= one;
        }
            }

      one <<= 1;
        }

    k = p+W;

    while (i++ < k)
        {
      bvc |= one;
      one <<= 1;
        }

    *b++ = bvc;
      }
  }

 for (a = 0; a < SIGMA; a++)
 {
   Pc[a] = TRAN[a][0];
 }

 Ebit = (((long) 1) << (W-1));
}

//———————————————————————————
//search()
//This is the bit-parallel approximate string matching function
//
//Parameters:
//char *filepath - path to data file
//long dif - maximum allowable string mismatches
//long sourceStartIndex - starting index within the data file for this search
//long sourceLen - length of characters to search from the start index
//long leftBoundPadSize - the number of characters padded on the left
//            boundary for this search
//int nodeId - the id of the processor performing this search
//
```

```
void search(   char *filepath,
        long dif,
        long sourceStartIndex,
        long sourceLen,
        long leftBoundPadSize,
        int nodeId )
{
  printf( "Node %i\n", nodeId );

  int num = 0;//container for the number of bytes read from file each time
  int i = 0, base = 0, diw = 0, a = 0, Cscore = 0;
  Scell *s, *sd;
  unsigned WORD pc, mc;
  register unsigned WORD *e;
  register unsigned WORD P, M, U, X, Y;
  Scell *S, *SE;

  const int BUF_SIZE = ( sourceLen >= BUF_MAX ) ? BUF_MAX : sourceLen;
  char *buf = (char*)malloc( BUF_SIZE );

  S  = (Scell *) malloc(sizeof(Scell)*seg);
  SE = S + (seg-1);

  diw = dif + W;

  sd = S + (dif-1)/W;
  for (s = S; s <= sd; s++)
  {
    s->P = All;
    s->M =  0;
    s->V = ((s-S)+1)*W;
  }

  //Open the source file
  FILE *ifile;
  ifile  = fopen( filepath, "r" );
  if( ifile == NULL )
  {
    printf( "Error: unable to open source file for reading.\n" );
    return;
  }

  //Move to starting index within source file
  fseek( ifile, sourceStartIndex, SEEK_SET );

  base = 1 - rem;
  long totalBytesRead = 0;
  long bytesLeft = 0;
  long mainLoopCounter = 0;
```

```
while( 1 )
{
  //reset the read buffer to zero before reading
  memset( buf, 0, BUF_SIZE );
  bytesLeft = ( sourceLen + leftBoundPadSize )- totalBytesRead;
  if( bytesLeft <= 0 )
    break;
  if( bytesLeft > BUF_MAX )
  {
    num = fread( buf, 1, BUF_MAX, ifile );
  }
  else
  {
    num = fread( buf, 1, bytesLeft, ifile );
  }

  if( totalBytesRead != 0 )
  {
    base += num;
  }

  totalBytesRead += num;

  if( num == 0 )
    break;

  i = 0;
  if (sd == S)
  {
    P = S->P;
    M = S->M;
    Cscore = S->V;
    for (; i < num; i++)
    {
      a = buf[i];

      U  = Pc[a];
      X  = (((U & P) + P) ^ P) | U;
      U |= M;

      Y = P;
      P = M |   (X | Y);
      M = Y & X;

      if (P & Ebit)
      {
        Cscore += 1;
      }
      else if (M & Ebit)
```

```
        {
          Cscore -= 1;
        }

        Y = P << 1;
        P = (M << 1) |   (U | Y);
        M = Y & U;

        if (Cscore <= dif)
        {
          break;
        }
            }

      S->P = P;
      S->M = M;
      S->V = Cscore;

      if (i >= num)
      {
        mainLoopCounter += 1;
        continue;
      }

      if (sd == SE)
      {
#ifdef SHOW_LOCATIONS
        if( mainLoopCounter == 0 )
        {
          printf("  Match type 1 at %d\n",
          (  base+i ) + ( sourceLen * nodeId  ) + num - leftBoundPadSize );
        }
        else
        {
          printf("  Match type 1 at %d\n",
          (  base+i ) + ( sourceLen * nodeId  ) - leftBoundPadSize );
        }
#endif
      }

      i += 1;
        }

    for (; i < num; i++)
    {
      e  = TRAN[buf[i]];
      pc = mc = 0;
      s  = S;
```

```
while (s <= sd)
{
  U  = *e++;
  P  = s->P;
  M  = s->M;

  Y  = U | mc;
  X  = (((Y & P) + P) ^ P) | Y;
  U |= M;

  Y = P;
  P = M |  (X | Y);
  M = Y & X;

  Y = (P << 1) | pc;
  s->P = (M << 1) | mc |  (U | Y);
  s->M = Y & U;

  U = s->V;
  pc = mc = 0;

  if (P & Ebit)
  {
    pc = 1; s->V = U+1;
  }
  else if (M & Ebit)
  {
    mc = 1; s->V = U-1;
  }

  s += 1;
}
if (U == dif && (*e & 0x1 | mc) && s <= SE)
{
  s->P = All;
  s->M = 0;

  if (pc == 1)
  {
    s->M = 0x1;
  }
  if (mc != 1)
  {
    s->P <<= 1;
  }

  s->V = U = diw-1;
  sd   = s;
}
```

```
      else
            {
        U = sd->V;

        while (U > diw)
                {
          U = (–sd)->V;
                }
            }

      if (sd == SE && U <= dif)
      {
#ifdef SHOW_LOCATIONS
        if( mainLoopCounter == 0 )
        {
          printf("  Match  type 2 at %d\n",
          (  base+i ) + ( sourceLen * nodeId  ) + num - leftBoundPadSize );
        }
        else
        {
          printf("  Match  type 2 at %d\n",
          (  base+i ) + ( sourceLen * nodeId  ) - leftBoundPadSize );
        }
#endif
      }
        }

    while (sd > S)
    {
      i = sd->V;
      P = sd->P;
      M = sd->M;
      Y = Ebit;

      for (X = 0; X < W; X++)
            {
        if (P & Y)
                {
          i -= 1;
          if (i <= dif)
          {
            break;
          }
                }
        else if (M & Y)
        {
          i += 1;
        }
        Y >>= 1;
          }

      if (i <= dif)
      {
        break;
      }

      sd -= 1;
        }

    mainLoopCounter += 1;
  }//end main while loop

  if (sd == SE)
  {
    P = sd->P;
    M = sd->M;
    U = sd->V;
    for (i = 0; i < rem; i++)
    {
      if (P & Ebit)
      {
        U -= 1;
      }
      else if (M & Ebit)
      {
        U += 1;
      }

      P <<= 1;
      M <<= 1;

      if (U <= dif)
      {
#ifdef SHOW_LOCATIONS
        if( mainLoopCounter <= 1 )
        {
          printf(" Match  type 3 at %d\n",
          (base+i) + ( sourceLen * nodeId ) + num - leftBoundPadSize  );
        }
        else
        {
          printf(" Match  type 3 at %d\n",
          (base+i) + ( sourceLen * nodeId - leftBoundPadSize ) );
        }
#endif
      }
        }
  }

 free( buf );
}


//————————————————————————————————
//The following functions are used to parse the input pattern
//that will be used for the search
//
int scan1(pat)
register char *pat;
{
 register int vlen;
 patlen = vlen = 0;
 while (*pat != '\0')
 {
  pat += 1;
  patlen += 1;
  vlen += 1;
 }

 return (vlen);
}

void scan2(pat,vlen) register char *pat; int vlen;
{
 register int c, comp;
 register char *vpt;

 vpt = (char *) malloc(vlen);
 patvec = (element *) malloc(sizeof(element)*patlen);

 patlen = 0;
 while( *pat != '\0' )
 {
  patvec[patlen].type = CHAR;
  patvec[patlen].value = vpt;
  *vpt++ = *pat++;
  patlen += 1;
 }
}

void encode_pattern(pat) char *pat;
{
 int vlen;
 vlen = scan1(pat);
 scan2(pat,vlen);
}
```

```
//——————————————————————————-
//main()
//This is the entry point for the program and initiates the search
//algorithm. The multiprocessor parallelization is achieved by
//segmenting the search data by dividing it by the number of processors
//involved in the operation. Each processor then individually performs
//a search with its subset of the search data and reports its findings
//individually.
//
int main(argc,argv) int argc; char *argv[];
{
  //MPI variables
  int nodeId;
  int nodeCount;

  //MPI Initialization
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &nodeId );
  MPI_Comm_size( MPI_COMM_WORLD, &nodeCount );

  //Local variables
  long dif = 0;
  char *pat = NULL;
  char *filepath = NULL;
  FILE *ifile = NULL;
  long fileSize = 0;
  long dataSubsetSize = 0;
  long targetStringSize = 0;
  struct timeval startTime, endTime;
  double timeElapsedInSeconds = 0.0;
  int procIndex = 0;

  //Check input arguments
  if( argc != 4 )
  {
    if( nodeId == CONTROL_PROCESS )
    {
      printf( "Invalid parameter set. Unable to start search.\n" );
      printf( "Usage: %s [ pat ] [ dif ] [ file ]\n", argv[0] );
    }

    goto FINALIZE;
  }

  pat = argv[1];
  dif = atoi(argv[2]);
  filepath = argv[3];
```

```
targetStringSize = strlen( pat );

//Attempt to open the source file that will be used to search
ifile = fopen( filepath, "r" );
if ( ifile == NULL )
{
  if( nodeId == CONTROL_PROCESS )
    printf( "Can't open file %s. Unable to start search.\n", filepath );

  goto FINALIZE;
}

//get the size of the file
fseek( ifile, 0L, SEEK_END );
fileSize = ftell( ifile );
fclose( ifile );

//calculate the data subset size that will be used for each processor
dataSubsetSize = fileSize / nodeCount;
if( dataSubsetSize <= 0 )
{
  if( nodeId == CONTROL_PROCESS )
    printf( "Insuficient data for all processor nodes. \
        Unable to start search.\n" );

  goto FINALIZE;
}

//check that the max differences allowed for matches
//is <= than the length of the pattern
if( dif > targetStringSize )
{
  if( nodeId == CONTROL_PROCESS )
    printf( "Invalid parameter for allowed mismatches, \
        cannot be larger than pattern size. Unable to start search.\n" );

  goto FINALIZE;
}

//search is ready to start, print out relevant parameters
if( nodeId == CONTROL_PROCESS )
{
  printf( "\n****************************************\n" );
  printf( "pat=%s\n", pat );
  printf( "pat length=%i\n", targetStringSize );
  printf( "dif=%i\n", dif );
  printf( "data file=%s\n", filepath );
  printf( "file size bytes=%i\n", fileSize );
  printf( "number of processors=%i\n", nodeCount );
  printf( "data subset size=%i\n", dataSubsetSize );
  printf( "****************************************\n\n" );
 }

 //Start the timing measurement
 gettimeofday(&startTime, NULL);

 //Encode the pattern that will be used in the search
 encode_pattern(pat);

 //setup the search
 setup_search();

 //Launch the search segments in each processor
 for( procIndex=CONTROL_PROCESS; procIndex < nodeCount; procIndex++ )
 {
  //Only do this for the current process instance
  if( nodeId == procIndex )
  {
   //get the starting index for the subset of this node.
   //Note that the leftmost subset needs no patching
   //all other nodes get an extra (targetSize -1) of search area at the left.
   long leftBoundPadSize = ( nodeId > CONTROL_PROCESS ) ? targetStringSize - 1 : 0;
   long dataStartIndex = ( dataSubsetSize * nodeId ) - leftBoundPadSize;

   //perform the actual search
   search(  filepath, //path to file containing the search string
     dif, //maximum allowable differences for a string match
     dataStartIndex,  //the starting index for this subset
         //within the data file
     dataSubsetSize,  //the size of the data subset
     leftBoundPadSize,  //the amount of padding on the
          //left of the data subset for this node
     nodeId ); //the identifier for this node

   gettimeofday(&endTime, NULL);//get end time
   //compute the time elapsed in seconds
   timeElapsedInSeconds = endTime.tv_sec - startTime.tv_sec
    + ( ( endTime.tv_usec - startTime.tv_usec ) / 1.e6 );

    //output the time taken to process LD for this node
   printf( "time in seconds=%.9f\n\n", timeElapsedInSeconds );
  }
 }

FINALIZE:
 //Cleanup
 MPI_Finalize();
```

```
  return 0;
}
```

REFERENCES

[1] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35:74–82, 1992.

[2] R. A. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Lecture Notes in Computer Science*, volume 1075, pages 1–23. Springer-verlag, New York, 1996.

[3] Mathieu Blanchette and Martin Tompa. Discovery of regulatory elements by a computational method for phylogenetic footprinting. *Genome Res*, 12(5):739–748, 2002.

[4] William S. Bradshaw and Richard D. Storey. *Biological Science, A Molecular Approach.* BSCS, 1990.

[5] Alan Filipski and Sudhir Kumar. *The Evolution of the Genome.* Burlington, MA : Elsevier Academic, 2005.

[6] Richard W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 2(26):147–160, 1950.

[7] Ross C. Hardison. Comparative genomics. *PLoS biology*, 1(2):58, 2003.

[8] Neil C. Jones and Pavel A. Pevzner. *An Introduction to Bioinformatics Algorithms.* MIT Press, 2004.

[9] Richard Karp and Michael Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[10] R. D. Knight and L. F. Landweber. Rhyme or reason: Rna-arginine interactions and the genetic code. *Chemistry And Biology*, 5(9):215–220, 1998.

[11] G.M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer Systems and Sciences*, 1(37):63–78, 1988.

[12] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[13] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31.

[14] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[15] Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, (6):132–137, 1985.

[16] James Watson and Francis Crick. Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 1953.

[17] Alden Wright. Approximate string matching using within-word parallelism. *Software - Practice and Experience*, 24(4):337–362, 1994.

[18] Sun Wu and Uri Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.

[19] Sun Wu, Uri Manber, and Gene Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15:50–67, 1996.

[20] Elena Zaslavsky and Mona Singh. A combinatorial optimization approach for diverse motif finding applications. *Algorithms Mol Biol.*, 1(13), 2006.