WAVEFRONT LONGEST COMMON SUBSEQUENCE ALGORITHM ON MULTICORE AND GPGPU PLATFORM

BILAL MAHMOUD ISSA SHEHABAT

UNIVERSITI SAINS MALAYSIA

2010

WAVE-FRONT LONGEST COMMON SUBSEQUENCE ALGORITHM ON MULTICORE AND GPGPU PLATFORM

By

BILAL MAHMOUD ISSA SHEHABAT

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

June 2010

CHAPTER 1

INTRODUCTION

1.1 Background

String comparison is a central operation in numerous applications. It has a critical task in many operations such as data mining, spelling error correction and molecular biology (Tan et al, 2007; Michailidis and Margaritis, 2000). String comparison aims to evaluate the similarity between a pair of given strings defined over a common finite alphabet (Michailidis and Margaritis, 2000). String comparison is used in spelling error correction, which tries to find a dictionary entry most resembles to a given word. In molecular biology, sequence comparison is used to find the homology between the bio-sequences (Tan et al, 2007; Michailidis and Margaritis, 2002). String matching has two paradigms; Exact matching and Approximate matching. The longest common subsequence (LCS) problem which is a very well known classical problem in computer science has a lot of applications. It is an approximate string matching problem and the simplest prototype of a sequence alignment algorithm. The longest common subsequence problem is an obvious measure for the closeness of two strings to find the maximum number of identical symbols between them taking into consideration the symbol order (Tan et al, 2007).

A subsequence of a given substring is any string obtained by deleting zero or more symbols from the given string. It is called as a common subsequence of two or more strings when it exists in both. The longest common subsequence is the common subsequence that has the maximum length (Strate and Wainwright, 1990; Giegerich et al, 2004).

1.2 Motivation

Longest common subsequence (LCS) problem is a very important problem used in various applications such as file comparison, word processing, molecular biology. Longest Common Subsequence has many implementations, among which one is based on dynamic programming (DP) solution. This solution gives an optimal result but takes a quadratic time and space complexities. While the time is an important factor, many researches have been done on the Dynamic Programming based Longest Common Subsequence to speed up its execution. One of these solutions is to run the LCS in parallel in order to reach the best execution time. Parallel programming is taking a new dimension, a new technology in order to reach the best execution speed. A new technology that uses the graphics hardware to implement different algorithms in parallel and run them on the Graphical Processing Unit (GPU) using a new platform called Compute Unified Device Architecture (CUDA) rather than uses the traditional parallel techniques those run on the Central Processing Unit (CPU).

1.3 Problem Statement

The basic implementation of the longest common subsequence algorithm consumes a quadratic time. The parallel method is used in order to reduce the execution time. The basic longest common subsequence algorithm has a high data dependency inhibits parallelism. The important question that needs to be asked is:*”how to make the design of the longest common subsequence able to be parallel, and will the parallel solution using the General Purpose Graphical Processing Unit (GPGPU) enhance the execution speed?”*

1.4 Research Objectives

The objectives of this research are:

- To propose a new design of the LCS problem using wave-front approach to eliminate the data dependency so that it can be parallelizable. Using wave-front approach.
- To design the proposed LCS problem (dynamic programming) for implementation on the GPGPU platform, using CUDA (Compute Unified Device Architecture) to improve its speed.
- To Implement the proposed design on Multicore platform using OpenMP.

1.5 Contributions

The expected contributions of this research are:

1. A new design of the basic implementation of the longest common subsequence problem to be parallel.

2. CUDA based parallel LCS on the GPGPU.

3. Parallel LCS on the Multi-core using OpenMP.

1.6 Scope

The scope of this study has 3 phases: (1) finding the longest common subsequence (LCS) of two strings using the wave-front approach. (2) Parallelization phase on multi-core CPU. (3) Parallelization phase using CUDA platform on the graphics hardware GPGPU.

This research focuses on the longest common subsequence algorithm and how to improve its execution speed depending on parallelization using different architectures.

1.7 Research methodology

This section shows the principles and methods of this research by discussing the parts of the research methodology such as research procedure, theoretical framework and research design.

1.7.1 Research Procedure

The first step of the procedure is to collect the data for the experiment by downloading it from the internet. The data used is a standard benchmark data being used in string matching applications. These types of data are DNA sequence and protein sequence.

The second step in the procedure is to change the design of the basic LCS algorithm into the wave-front approach, in order to reduce the data dependency and make the LCS algorithms able to run in parallel, there will be a comparison between the basic and the updated designs of the LCS problem during this step.

The third and last step is to parallelize the wavefront LCS using the multicore and the graphics hardware and compare the parallel results in the two parallel implementations. Figure 1.1 explains the steps of the research procedure.

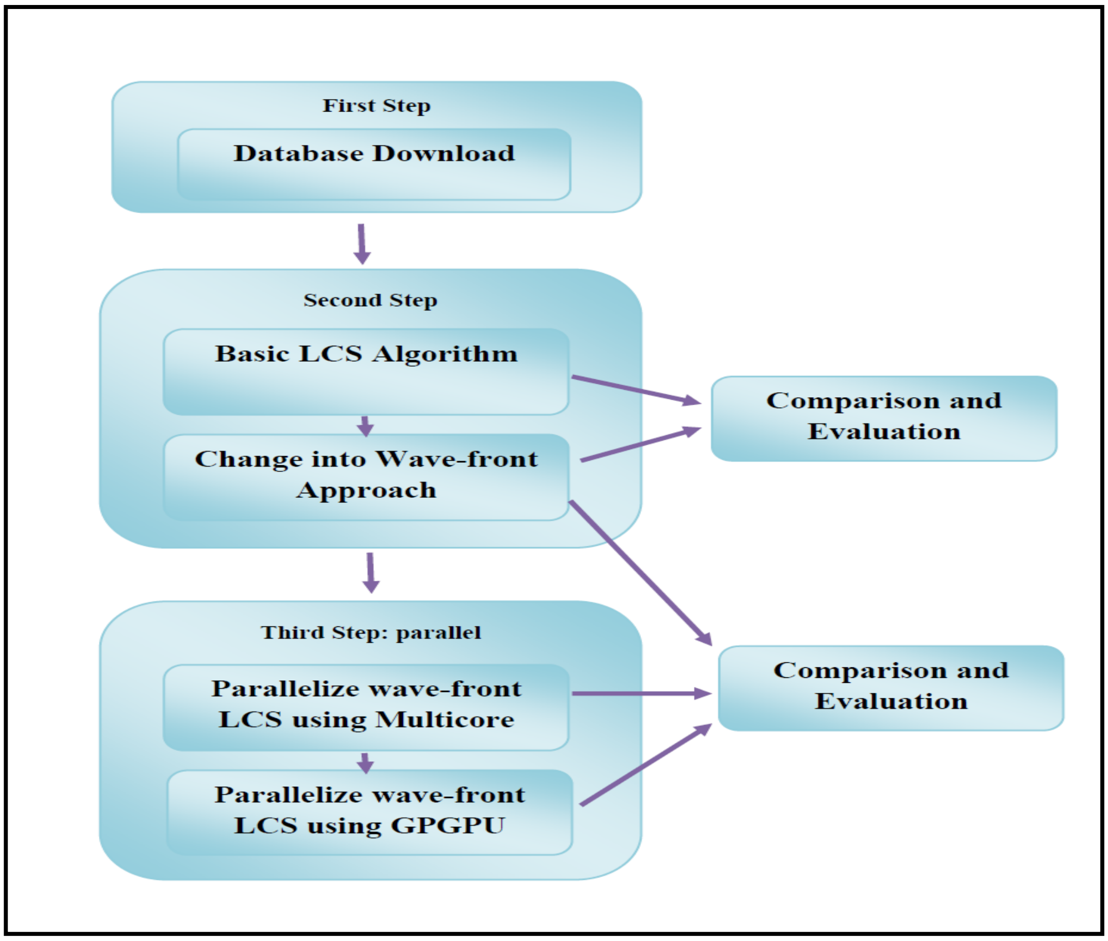
First Step
Database Download
Second Step
Basic LCS Algorithm
Change into Wave-front Approach
Comparison and Evaluation
Third Step: parallel
Parallelize wave-front LCS using Multicore
Parallelize wave-front LCS using GPGPU
Comparison and Evaluation

Figure 1.1: Research Procedure

1.7.2 Theoretical Framework

Lots of studies have been done on the LCS problem, some are concerned with improving the space complexity, and some are concerned with improving the time complexity. The intent of using parallel platforms to run the algorithms is to get a high performance and execution speed. Some experiments to enhance the LCS algorithm are discussed in the related work section in this research. New dimensions have appeared in parallel computing area such as exploiting the graphics hardware to run non-graphical (general purpose) algorithms. Hence the starting point of our research is taken.

1.7.3 Research Design

There are some attributes and variables involved in the research design such as a purpose of the study, type of investigation, study setting and time horizon. The purpose of the study in this research is a "Case Study Analysis" since the method is qualitative in nature. The type of investigation is "causal" seeing that we need to change something in the algorithm itself. Since our study is showed doing some change to an existing algorithm in order to make it able to be parallel. The setting of our study is a lab experiment. And the time horizon of the study is a "Cross Sectional" since the data is collected only once and used as a standard in all the experiments without needing to collect data for different situations.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter discusses the dynamic programming technique, general information about the string matching, string matching groups and algorithms. Furthermore, this chapter discusses some of the experiments related to our work in this thesis. The parallelism is discussed including parallelism types and parallel models.

2.2 Dynamic Programming

Dynamic programming (DP) is a classical powerful and well-known technique for solving large kinds of optimization problems (Tan and Sun and Gao, 2007; Giegerich et al**,** 2004). The programming in this context doesn't mean the computer programming; it is a tabular method of solving the problems.

Divide and Conquer technique solves the problem by dividing it into smaller sub-problems, each of one can be solved independently. These sub-problems are in turn recursively divided into smaller sub-problems and solved independently and so on **(**Strate and Wainwright, 1990**).** In contrast, dynamic programming is applicable when the sub-problems are not independent (Cormen et al, 2001)**.** In general Dynamic programming technique can be thought of as the divide and conquer principle taken to an extreme **(**Strate and Wainwright, 1990**).** The essence of dynamic programming algorithms is that they trade space for time by storing solutions to sub-problems rather than recomputing them **(**Strate and Wainwright, 1990).

Dynamic programming can be applied to the optimization problem if it has an optimal substructure and overlapping sub-problems. To solve an optimization algorithm using dynamic programming, two things must be done. First is to characterize the structure of the optimal solution, the optimal solution of the problem comes from the optimal solutions of the sub-problems. Second the overlapping sub-problems. The optimization algorithm has overlapping sub-problems when the algorithm solves the same sub-problems over and over rather than generating new sub-problems. The benefit of overlapping sub-problems is that the solution of the sub-problem can be stored in a table for use when needed (Morrison, 1997). Using the dynamic programming to solve a problem needs the following steps (Eddy, 2004).

- Define the optimal structure of the solution, depending on a scoring system to find the general definition (formula) of the problem.
- Filling the dynamic programming matrix, saving the optimal solution of sub-problems, in this case each sub-problem will be solved only once rather than the simple recursion.
- Calculating the optimal score using bottom up approach (from the smallest sub-problems to progressively bigger sub-problems).
- Trace-back the matrix to extract the result, this step may need an extra information to be stored in the dynamic programming matrix, this step starts from the cell (M,N) and follows the appropriate path depending on the formula which is determined in the first step until reaching cell(0,0).

2.3 String Matching Groups

String matching is a technique to compare two or more strings to find if they are similar or not. It takes a part in many computer science applications as data processing, speech recognition, information retrieval, search engines on the internet, vision for two dimensional image recognition and computational biology (Michailidis and Margaritis, 2000; Michailidis and Margaritis, 2002).

String matching problem consists of two parts, which are text and pattern, where the text is larger in size than the pattern. The matching is done by attempting to find identical characters between the text and pattern. Many algorithms have been studied to speedup the matching process (Michailidis and Margaritis, 2000) (Michailidis and Margaritis, 2002).String matching has two paradigms, which are the exact string matching and the approximate string matching as shown in Figure 2.1.

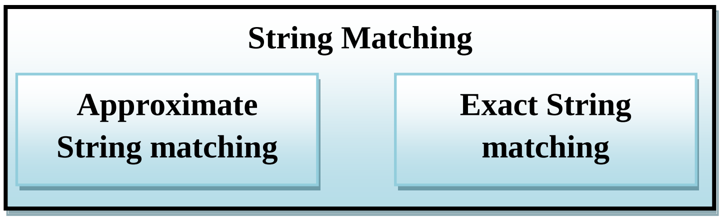

Figure 2.1 String matching algorithm types

2.4 Exact String Matching

String matching consists of finding one or more generally all the occurrences of a short pattern $P=P_0P_1.....P_{m-1}$ of length m in a large text $T=T_0T_1...T_{n-1}$ of length n, where $m,n>0$ and $m\leq n$. Both P and T are built over the same alphabet (Michailidis and and Margaritis 2000). As shown in Figure 2.2, exact string matching has a four types.

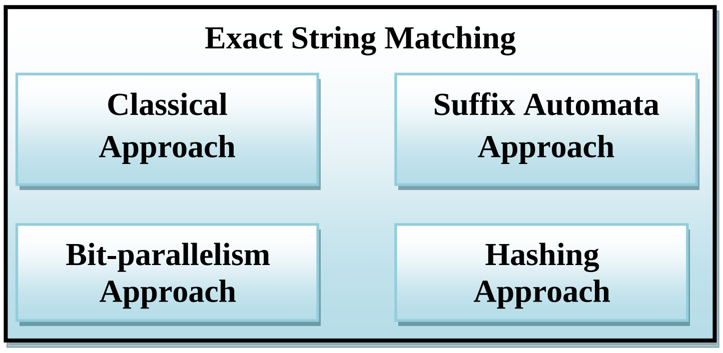

Figure 2.2 Exact String Matching Algorithm Types

2.4.1 Classical Algorithms

The classical approach exact string matching algorithms are based on character comparisons. Many algorithms use this approach like Brute-Force (BF) algorithm, The Knuth-Morris-Pratt (KMP) algorithm, the Boyer-Moore (BM) algorithm.

The simplest algorithm is Brute-Force (BF) algorithm, which has no preprocessing phase and performs the comparison from the left to the right. It has *O(mn)* time complexity in the worst-case (Michailidis and Margaritis, 2000). The Knuth-Morris-Pratt (KMP) algorithm performs the comparison from the left to the right. KMP has a preprocessing phase which takes *O(m)* time and space, and is considered as the first discovered algorithm that has a linear time (Michailidis and Margaritis, 2000). Boyer-Moore (BM) algorithm performs the comparison of the characters in the text and the pattern from the right to the left. BM algorithm uses two heuristics called occurrence heuristic and match heuristic, the maximum shift of these two heuristics is the length of the character shift when the mismatch happens or after the complete match. *O(m+|Ò|)* is the processing time and space of the two heuristics. BM takes *O(n+rm)* searching phase time in its worst-case, *r* here means the number of how many time the pattern occurs in the text (Michailidis and Margaritis, 2000).

2.4.2 Suffix Automata Approach

Suffix automaton also called DAWG (Deterministic Acyclic Word Graph) on String *S* is the minimal deterministic finite automaton that recognizes all the substrings of *S*.

The Reverse Factor (RF) algorithm uses the smallest suffix automaton of the reverse pattern to perform the text from right to left (Michailidis and Margaritis, 2000). RF algorithm requires a linear time in the preprocessing phase and space in the length of the pattern. The searching phase of RF algorithm has an optimal time complexity in the average case and quadratic time, in the worst case. It performs *O (nlogm/m)* comparisons between characters on the average (Michailidis and Margaritis, 2000).

2.4.3 Bit-Parallelism Approach

Bit-parallelism is a technique aims to speedup the matching process using bit parallelism operations by cutting down the number of bits in the computer word (Navarro, 2001). This approach has two main advantages, first: simplicity, where the preprocessing and searching phases are very simple. Second advantage is the flexibility, where one text character is processed by a constant time and delay and no buffering and the text does not need to be stored (Michailidis and Margaritis, 2000).

Shift-Or (SO) algorithm is a bit-wise technique algorithm, it creates a mask in the preprocessing phase for every character in the alphabet. Searching the characters in the string is directed from left to right, retrieving the mask of the character which is being read. It uses a variable R to keep track of the characters (Leidig and trefftz, 2007).

2.4.4 Hashing Approach

Hashing aims to avoid the quadratic number of the character comparisons in most practical situations (Charras and Lecroq, 2004). Karp-Rabin (KR) algorithm computes the signature or the hashing function of each possible M-character substring in the text and checks its equality with the hashing function of the pattern. Karp-Rabin algorithm has *O(m)* preprocessing phase and *O(mn)* searching phase(Michailidis and Margaritis, 2000).

2.5 Approximate String Matching

The difference between the exact string matching and the approximate string matching is that the exact string matching searches for a complete identification between the pattern with a substring in the text. While the approximate string matching focus on finding a similarity between the pattern and a substring inside the text (Michailidis and Margaritis, 2000;Giegerich et al, 2004). The string matching algorithm can be on-line that is the text is not known in advanced and needs a preprocessing phase or off-line which means no need to a preprocessing phase (Michailidis and Margaritis, 2002).

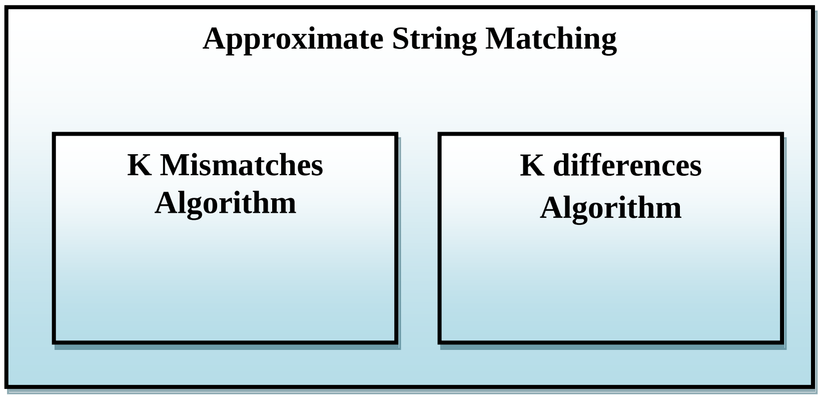

Figure 2.3: Approximate String Matching Algorithm Types

There are two classes of the approximate string searching: string searching with *k* mismatches and string searching with *k* differences as shown in Figure 2.3. Two well known distance functions represent these two classes. The hamming distance represents the string searching with *k* mismatches, where the hamming distance shows how many mismatched characters in two equal length strings. The Levenshtein distance represents the minimum number of character insertions, deletions and substitutions which are needed to transmute one string to another. Taking into consideration that the two strings are not important having the same length. It is referred as string searching with *k* differences (Michailidis and Margaritis, 2000).

2.5.1 String Matching with *K* Mismatches

The searching phase has four approaches; classical algorithms, deterministic finite automata algorithms, counting algorithms and bit-parallelism algorithms. As shown in Figure 2.4, the searching phase for the string searching with *k* mismatches problems has four categories:

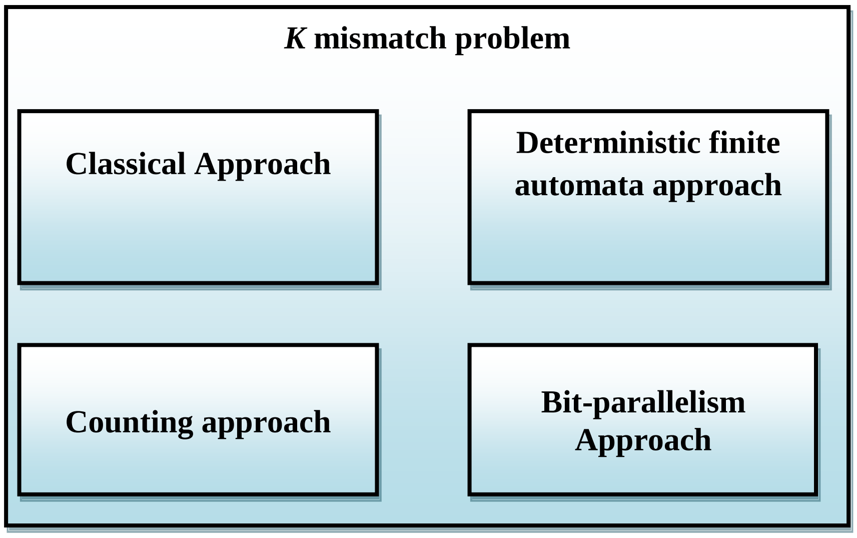

Figure 2.4: *K* Mismatch Approximate String Matching Types

(A) Classical Approach

When the string searching algorithms mainly rely on the character comparisons then it is called a classical string searching algorithms (Michailidis and Margaritis, 2002).

The Brute-Force (BF) algorithm has *O(mn)* time complexity in its worst case. It counts the number of mismatches happened during the left to right comparison between the complete pattern with the text substring, noticing that the preprocessing phase is not needed for this algorithm.

The first efficient developed algorithm is the Lindau-Vishkin (LV) algorithm. LV algorithm has a preprocessing phase that extracts information to decrease the required character comparisons during the searching phase; this algorithm takes *O(km log m)* time for preprocessing phase and *O(kn)* for searching phase. In spite of the LV algorithm is efficient it has a disadvantage that it requires extra space *O(k(m+n)),* which is not acceptable for practical purposes (Michailidis and Margaritis, 2002).

Tarhio-Ukkonen (TU) algorithm is based on Boyer-Moore-Harspool (BMH) exact searching algorithm, the TU algorithm has $O(m+k|\Sigma|)$ time and $O(k|\Sigma|)$ space as shown in Table 2.1 (Michailidis and Margaritis, 2002).

(B) Counting Approach

Arithmetic operations are used instead of character comparisons in the classical approach. Baeza-Yates-Perlberg (BYP) algorithm is a very practical and simple solution to the string searching with *k* mismatches problem and whose performance is independent on *k*, the worst case happens when all characters in *P* are distinct (Michailidis and Margaritis, 2002).

(C) Bit-Parallelism Approach

A common technique was found by Baeza-Yates and Gonnet that they considered every element in the pattern as a set of symbols rather than one symbol (Bayeza-yates, 1992).

The goal of taking the character as symbols (bits) is to perform many operations in parallel. This approach has many advantages such as simplicity, flexibility, and no buffering. Like Shift-Or (SO) algorithm which is an a bit-parallelism algorithms (Michailidis and Margaritis, 2002).

(D) Deterministic Finite Automata Approach

This kind of algorithms has an advantage where it can repeat the searching process during the matching operation (Zhang, 2003).

Table 2.1: Time and Space Complexities for String Matching with *K* Mismatches (Michailidis, 2002)

Algorithm	**Worst Case**	**Average Case**	**Preprocessing Time**	**Extra Space**
BF	Mn	Kn	-	1
LV	Kn	Kn	$Km \log m$	Km
TU	Mn	$kn(k/\|\Sigma\|+1/m-k)$	$m+ k\|\Sigma\|$	$k\|\Sigma\|$
BYP	N	$(1+m/\|\Sigma\|)n$	$2m+\|\Sigma\|$	$m+\|\Sigma\|$
SO	$mn \log k/w$	$mn \log k/w$	$(\|\Sigma\|+m) \log k/w$	$\|\Sigma\|+m \log k/w$

2.5.2 String Matching with *K* Differences

This type can be sorted into four approaches, which are the dynamic programming approach, filtering approach, deterministic finite automata approach and bit-parallelism approach as shown in Figure 2.5.

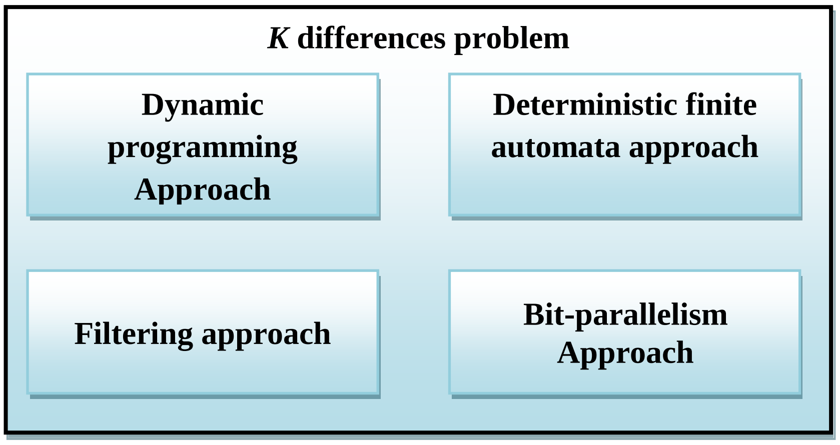
K differences problem
Dynamic programming Approach
Deterministic finite automata approach
Filtering approach
Bit-parallelism Approach

Figure 2.5: *K* Differences Approximate String Matching Types

(A) Dynamic Programming Approach

This approach is a classical solution to compute the edit distance between two strings, was found by Wanger and Fischer. Dynamic programming works as an accumulating process until it reaches the result (Michailidis and Margaritis, 2002).

SEL algorithm has O(mn) worst and average case running time. It is a search algorithm finds all approximate occurrences of the pattern string P in the text string T. Dynamic programming paradigm has utilized this algorithm in order to compute kn rather than mn entries (Michailidis and Margaritis, 2002). CUTOFF algorithm found by (Ukkonen, 1985), computes only a part of the dynamic programming array enhancing the execution time into O (nk).

New diagonal transition algorithms were developed based on computing the values in the incremented diagonal positions in the dynamic programming array. The Galil-Park (GP) algorithm based on diagonal transition takes O(m^2) time for preprocessing phase and O(nk) searching phase in the average or worse case (Michailidis and Margaritis 2002). The dynamic programming approach also adapted using "column partition approach" in order to increase the speed of the running time, as like the Chang-Lampe (CL) algorithm which is based on this approach (Michailidis and Margaritis, 2002).

(B) Deterministic Finite Automata Approach

The goal of this approach is to convert the general automaton into a deterministic one to reduce the states and memory requirements. (Ukkonen, 1985) proposed an algorithm in the kind of deterministic finite automaton (DFA), but it may take large time and space. Sometimes it requires large time and space requirements because of the large number of the generated states, which makes this algorithm insufficient (Michailidis and Margaritis, 2002).

(C) Filtering Approach

It is a newer method uses dynamic programming approach to drop the areas that cannot match in the text then apply another algorithm on it. For filtering the text a new algorithm based on Boyer-Moore-Harspool called TUD which has been found by Tarhio-Ukkonen (Tarhio and Ukkonen, 1993). COUNT is a new filtering algorithm found by Navarro is based on counting the matching positions in the pattern and the text depending on the k differences (Navarro, 2001). Pattern partition approach is a simple filter proposed by (Wu and Manber, 1992) it can conclude that there is matching between a substring in the text with a substring in the pattern if an occurrence with at most k differences of the pattern happen (Michailidis and Margaritis, 2002). BYPEP is a new suggested algorithm by (Baeza-Yates, 1992), combines the pattern partition approach with the traditional multiple string matching searching algorithms (Michailidis and Margaritis, 2002).

(D) Bit-Parallelism Approach

This approach can be applied to the parallelization of the nondeterministic finite automata (NFA) and the parallelization of the dynamic programming array. This approach has been used by Wu and Manber (WM) to simulate the automaton by rows. BYN algorithm uses the bit parallelism to parallelize the NFA. Another algorithm called Myers (MYE) has an optimal speedup uses a bit parallel simulation of the dynamic programming array (Michailidis and Margaritis, 2002).Table 2.2 shows the time and space complexities of some string matching algorithms with k differences.

Table 2.2: Time and Space Complexities of String Matching with *K* Differences (Michailidis 2002).

Algorithm	Worst case	Average case	Preprocessing time	Extra space
SEL	Mn	Mn	-	Mn
CUTOFF	Mn	Kn	-	M
GP	Kn	Kn	m^2	m^2
CL	Mn	$\mathrm{kn}/\sqrt{\lvert\Sigma\rvert}$	$M\lvert\boldsymbol{\Sigma}\rvert$	$M\lvert\boldsymbol{\Sigma}\rvert$
TUD	mn/k	$(\lvert\Sigma\rvert/\lvert\Sigma\rvert-2k)$ $kn(k/\lvert\Sigma\rvert+2k^2+l/m)$	$(k+\lvert\Sigma\rvert)m$	$m\lvert\Sigma\rvert$
COUNT	Mn	N	$\lvert\Sigma\rvert+m$	$\lvert\Sigma\rvert$
BYPEP	-	$n,k\leq m/\log n$	M	m^2
WM	$kn[m/w]$	$kn[m/w]$	$M\lvert\Sigma\rvert+k[m/w]$	$m\lvert\Sigma\rvert$
BYN	N	N	$\lvert\Sigma\rvert+m\min(m, \lvert\Sigma\rvert)$	$\lvert\Sigma\rvert$
MYE	mn/w	kn/w	$m\lvert\Sigma\rvert$	$\lvert\Sigma\rvert$

2.6 Edit Distance Solution

The distance between two strings *x* and *y* defined as the minimal cost of sequence operations to transform *x* into *y*. However, there are four possible operations; insertion, deletion, substitution or replacement and transposition. Edit distance allows to insert, delete, substitute simple characters in both strings, there are two types of edit distance; when the operations have different cost or depend on the involved characters then it is called *general edit distance*, if the all operations cost 1 then it called *simple edit distance* or *edit distance* (Navarro, 2001).

There exist many distance functions such as levenshtein distance, hamming distance, episode distance and longest common subsequence distance. However, levenshtein distance is symmetric allows a minimal number of insertions, deletions, and substitutions to make two strings equal. The search problem in many cases called string matching with *k* differences. Hamming distance also considered to be a symmetric, search problem in many cases called string matching with *k* mismatches. It allows only substitutions, which cost one, on the other hand, is episode distance, which is considered as asymmetric, which allows only insertions that cost 1, and it may not be possible to convert *x* into *y*, in many cases called episode matching. Longest common subsequence distance allows insertions and deletions that have the cost 1. This distance measures the length of the longest pairing of characters that can be made between both strings, and this distance is symmetric (Navarro, 2001).

2.7 Related Work

In this section, we are summarizing some experiments that are related to the longest common subsequence problem. We mentioned three kinds of experiments;

enhancements done on the longest common subsequence algorithm, parallelism done on the longest common subsequence problem and parallel with graphics hardware.

2.7.1 Enhancements on LCS

This section mentions some enhancements done to the longest common subsequence algorithms. The enhancements are focusing to improve the time and space complexity to the algorithm.

(A) SB_LCS (Stack Based)

While LCS algorithm takes a large space complexity which is the multiplication of the sequence lengths, e.g. two sequences with 50kb needed memory 50kb * 50kb, which is not applicable to ordinary computers. A solution has been proposed to reduce the memory complexity in the LCS algorithm at a forward path.

The proposed algorithm called SB_LCS (Stack Based LCS), saves the information in a stack if it cannot be reproduced at the backward path. This method increases the input DNA sequence several times. SB_LCS can test DNA with length up to 100 kb, and have a time complexity same as the basic LCS algorithm *O(mn)*(parvinnia et al, 2008).

(B) Bit-Vector LCS Algorithm

A proposed algorithm by (Crochemore et al, 2001) uses the bit-parallelism to get higher speed of the LCS. The proposed algorithm determines the length *p* of a longest common subsequence in *O(nm/w)* time complexity and *O(m/w)* space complexity, where w is the number of bits in the machine word.

(C) Fast Algorithm for LCS

The work proposed by (Hunt and Szymaski, 1977) presented an algorithm that has $O((r+n) \log n)$ running time, where r is the total number of ordered pairs of positions at which the two sequences match. While many algorithms are $O(n^2)$ worst case time complexity, this algorithm has $O(n^2 \log n)$ worst case time complexity. This algorithm exhibits an $O(n \log n)$ time complexity in a large number of applications when r is expected to be close to n.

2.7.2 Parallel Algorithms for LCS

This section mentions some proposed parallel solutions done to the longest common subsequence algorithms. Parallel algorithms are to improve the execution time of the algorithm regardless of the time and space complexity.

(A) FAST_LCS

(Liu et al, 2006) has presented a fast parallel implementation for the LCS problem called FAST_LCS, the main idea of this algorithm is to generate pairs of successors through successor tables using skipping and pruning techniques. This algorithm has two main phases, first is to search all identical character pairs and their levels, second is to trace back from the identical character pair at the largest level to obtain the longest common subsequence. FAST_LCS algorithm is faster than the basic Waterman algorithm, the required memory is max$\{4*(n+1)+4*(m+1),L\}$, L here is the number of identical character pairs , n is the length of the sequence X , m is the length of the sequence Y, and the time complexity of the parallel implementation is $O(|LCS(X,Y)|)$, where $|LCS(X,Y)|$ is the length of LCS of X,Y.

(B) RLE_LCS

(Freschi and Bogliolo, 2004) have proposed a new method to solve the LCS problem taking the advantage of RLE (run-length-encoded) to achieve better speed and improve the parallelism. RLE is a string compression technique represents the string as a sequence of runs instead of sequence of characters, e.g. string "acccttgggg" can be represented as "1a,3c,2t,4g" reaching the number of elements from 10 characters to 4 runs. The proposed algorithm achieves complexity $O(mN+Mn-mn)$, where M and N are the lengths of the original strings and m and n is the number of runs in their RLE representation.

(C) Cache-Oblivious LCS Using Graphics Hardware

The work by (Kloetzli et al, 2008) have proposed a solution of the longest common subsequence problem using the GPU, they identified a parallel memory access pattern that divides the problem into sub-algorithms and matches them to the multiple layers on parallel hardware, using a mix of the theoretical and experimental data including knowledge of the specific structure of the hardware and memory of each layer.

The developed method accelerates the cache oblivious method proposed by (Chowdhury et al, 2006) by solving sub-problems on the GPU. The advantage of this approach is that any algorithm that maps well onto the GPU can be used instead of being limited to a specific algorithm. Table 2.3 shows a comparison of the time and space complexity between the experiments mentioned in the related work.

Table 2.3: Comparison of the Related Experiments

Algorithm	Time Complexity	Space Complexity
Stack Based LCS	O(mn)	Not available
Bit Vector LCS	O(mn/w)	O(mn/w)
Fast LCS algorithm	O((r+n) log n) O(n^2 log n),worst case	Not available
FAST_LCS	O(|LCS(x,y)|)	Max{4*(n+1)+4*(m+1),L}
RLE_LCS	O(mN+Mn-mn)	Not available

2.8 Parallelism

Parallelism is the method that can carry out the huge and complex tasks faster. Another description of parallelism is the collaborative processors of computers that can solve the computational problems. It has strong relationship with life activities, such as parallel databases and data mining, web search engines, medical issues, industrial technology, multimedia technologies and others (Abdulrozaq, 2009).

2.8.1 Parallelism Types

There are two well known types of parallelism, according to the way of partitioning of data and functions.

(A) Data Decomposition

It is also called data parallelism or partitioning, it focuses on executing the same function, distributing the data across different parallel computing nodes. I can be static where each process has priority or dynamic where the subunits are specified to do some processes when free.

(B) Function Decomposition

Moreover, known as task parallelism, focus on executing many different functions on multiple cores.

2.8.2 Parallel Programming Models

There are two types of models shared memory model and distributed memory model. In the shared memory model, a group of processors are communicated with each other sharing the same memory. In distributed memory the connected processors, each one has its own memory that cannot be accessed by another processor. The common parallel platforms used are POSIX Thread, OpenMP, MPI and CUDA (Abdulrozaq, 2009).

(A) POSIX Threads (Pthread) (Portable Operating System Interface)

POSIX is a standardized programming interface to make the programming with threads easier. Using the shared memory and divides the problem into sub-problems. Pthread in C language has three elements: data type that refers to the thread, thread manipulation routines that refer to the library such as creation and initialization of the thread, the third element is the synchronization of the processors (Abdulrozaq, 2009).

(B) OpenMP

Supports multiplatform shared memory multiprocessing in C, C++ and Fortran on many architectures. It composed of libraries, compiler directives. OpenMP has