1992

# Fast parallel algorithms for approximate string matching

Yi Jiang
*The University of Montana*

# Fast Parallel Algorithms for Approximate String Matching

Yi Jiang
Computer Science, Univ. of Montana

For the degree of
Master of Science, University of Montana

Fall, 1992

Approved by [illegible]

---------------------------
Chairman, Board of Examiners

[illegible]

---------------------------
Dean, Graduate School

December 10, 1992

---------------------------
Date

Jiang, Yi, M.Sci., Fall 1992 Computer Science

Fast Parallel Algorithms for Approximate String Matching

Director: Alden H. Wright [illegible]

## ABSTRACT

Given a text string, a much shorter pattern string, and an integer $k$, parallel algorithms for finding all occurrences of the pattern string in the text string with at most $k$ differences (as defined by edit distance) are discussed. First, a real-time parallel algorithm, which could be implemented on a systolic array using $m$ (the length of the pattern string) very simple processing elements, is proposed. After the algorithm gets started, it outputs the minimum edit distance from the pattern string to a substring of the text string at each time step. Thus, the algorithm is well-suited for real-time searching of text databases or biological nucleic acid sequence databases. Second, several different ways for solving the same problem with different CRCW-PRAM assumptions (*priority* model, *combination* model, and $common - value$ model) are developed. This class of algorithms uses $O(m \times n)$ or $O(m \times m \times n)$ processors and achieve a time complexity of $O(k)$.

*Key words.* approximate string matching, edit distance, systolic computation, CRCW-PRAM models.

**Contents**

**1 Introduction** **1**

**2 Related work and concepts** **3**

**3 A new systolic algorithm** **29**
- 3.1 Motivation of the new algorithm . . . . . . 29
- 3.2 The new algorithm . . . . . . . . . . . . . . 29
- 3.3 Complexity analysis . . . . . . . . . . . . 33
- 3.4 A further simplification of the processor complexity . . . . . . . . . . . . . . . . 33
- 3.5 Discussion . . . . . . . . . . . . . . . . . 39
- 3.6 Architectural Alternatives - message-passage P E network . . . . . . . . . . . . . . . . 40

**4 An** O(k) **CRCW algorithm and related ideas** **40**
- 4.1 Introduction. . . . . . . . . . . . . . . . . 40
- 4.2 O(k) time with O(m x n) processors. . . . . 44
- 4.3 O(k) time with O(n x m x m) processors. . . 47
- 4.4 Conclusions . . . . . . . . . . . . . . . . . 47

**5 Acknowledgement** **48**

## 1 Introduction

In many branches of scientific research, it is often necessary to search a large database for approximate occurrences of an interesting pattern. This is especially important in molecular biology, and information retrieval in general. In the case of molecular biology, the need for development of advanced algorithms and technologies in this and related areas is so urgent that a special article[Lander91] was published in November issues of both *Communication ACM* and *IEEE Computer*, inviting computer scientists to join forces with molecular biologists in the human genome project, one of the most promising and ambitious research projects in this century.

In existing nucleic acid databases, new data are being added at a rapid rate. The number of possible relationships between various nucleic acid sequences is significantly larger than the number of sequences. Furthermore, along with the rapid advances of molecular biology, the number of features of interest is also increasing. These trends do not show any signs of abating. As a result of all these, searching these databases for approximate occurrences of some given pattern becomes frequently necessary and difficult.

This thesis is concerned with the problem of finding all approximate matches of a pattern string $A$ of length $m$ in a much longer string $T$ of length $n$ ($n \gg m$). Approximate matching is defined in terms of edit distance, which allows for substitution, insertion, and deletion operations to transform $A$ into the substring of $T$.

The *edit distance* between strings $A$ and $B$, $d(A, B)$, is defined as the minimum number of edit steps in converting $A$ to $B$ using the following three kinds of edit steps:

(1) Delete an element from $A$.
(2) Insert an element into $A$.
(3) Replace an element of $A$ with another element.

*Example 2.1.* The edit distance between string A = "ADGTF" and string B = "AGCF" can be illustrated as follows:

```
A   D   G   T   F

A   _   G   C   F
```

Here, in converting A to B, element D in A is deleted, and element T is replaced by C. Thus, the edit distance between A and B is two (assuming the cost associated with any edit step to be 1).

The edit distance can be generalized by putting different costs on the operations or on the elements of the alphabet.

The *k-differences string-matching problem* is to find all occurrences of substrings of $T$ whose edit distance from $A$ is less than $k$ (When $k$ equals 0, this is the traditional exact string matching problem).

The systolic algorithm presented in this thesis is based on the dynamic programming method. It uses $m$ simple processing elements to solve the *k-differences string-matching* problem in $O(n)$ time. The processing elements are organized into a one-dimensional systolic array where each processing element communicates with at most 2 neighbors. The algorithm is real-time in the sense that the algorithm uses $n + m - 1$ time steps, and in each time step after the $m^{th}$, a result is output. This result is the minimum edit distance of the pattern string $A$ from a substring in the text string which ends at the right edge of a window. The right edge of the window slides from the leftmost position of the text string to the rightmost position, with its left edge being always at the left end of the text string.

The algorithm can easily be viewed as an EREW-PRAM algorithm.

In the study and research of parallel computing, abstract, theoretical computing models are often used in the literature. One important model is the parallel random-access machine, or PRAM in short.

In this thesis, several different ways of achieving a time complexity of $O(k)$ for solving the same problem with different CRCW-

PRAM assumptions are also presented.

Throughout this thesis, we use $A_i$ to denote the $i^{th}$ element of string $A$, and we use $A_i^j$ to denote substring $A_i A_{i+1} \dots A_j$, and use $d(A,B)$ to denote the edit distance between two strings $A$ and $B$.

## 2 Related work and concepts

Extensive research work has been done on this and related topics over the past decades. Sankoff, et. al. [Sankoff83] contains very extensive descriptions of the development of various dynamic programming algorithms as well as the background for scientific applications of string matching problems.

Historically, the development of basic dynamic programming methods to compute edit distance between two strings began in biology with Needleman and Wunsch [Needleman70], and was introduced independently by Sellers [Sellers74], and Wagner and Fischer [Wagner74]. Also see [Manber89] and [Sankoff83].

In this section, we'll describe basic dynamic programming methods as well as several representative (parallel and serial) algorithms to solve the *k-differences string-matching* problem.

Dynamic programming is a powerful and rigorous method to compute edit distance $d(A,B)$. The basic version of dynamic programming method used to compute $d(A,B)$ is described in several books on computer algorithms. Since it forms the basis of later developments and our new algorithm, we give a brief review of it here.

For simplicity, we assign a cost of 1 to each of the 3 edit steps: deletion, insertion, substitution.

Suppose we have two strings $A$ and $B$, of lengths $m$ and $n$, respectively. Define $C[i,j]$ ( $0 \le i \le m$, $0 \le j \le n$ ) to be the edit distance from $A_1^i$ to $B_1^j$. For $i \le m$ and $j \le n$, one can observe that $C[i,j]$ is either

(i) $d(A_1^{i-1}, B_1^{j-1}) + d(A_i, B_j)$, or
(ii) $d(A_1^i, B_1^{j-1}) + 1$ ( i.e., insert $B_j$ ), or
(iii) $d(A_1^{i-1}, B_1^j) + 1$ ( i.e., delete $A_i$ ).

Thus, we have

$$C[i,j] = \min(C[i-1,j]+1, C[i,j-1]+1, C[i-1,j-1]+c_{i,j}) \quad (1)$$

where $c_{i,j}$ denotes $d(A_i, B_j)$. In other words, $c_{i,j} = 0$ if $A_i = B_j$, and $c_{i,j} = 1$ otherwise.

Initially, $C[i,0] = i$ for $0 \le i \le m$ because $A_1^i$ differs from the empty text by $i$ insertion steps. Similarly, $C[0,j] = j$ for $0 \le j \le n$.

The following algorithm computes the edit distance $d(A,B)$.

*Algorithm* 0:

```
Initialization:
    for $0 \le i \le m$ do $C[i,0] \leftarrow i$
    for $0 \le j \le n$ do $C[0,j] \leftarrow j$
Main loop:
    for $i = 1$ to $m$ do
      for $j = 1$ to $n$ do
          $x \leftarrow C[i-1,j]+1$
          $y \leftarrow C[i,j-1]+1$
          if $A_i = B_j$ then
             $z \leftarrow C[i-1,j-1]$
          else
             $z \leftarrow C[i-1,j-1]+1$
          $C[i,j] \leftarrow \min(x,y,z)$
```

$C[m,n]$ is the output of this algorithm.

Thus, the essence of the dynamic programming method for this problem is to fill out a $m \times n$ matrix $C[0..m, 0..n]$, using the basic formula (1) iteratively.

*Example 2.2.* Applying algorithm 0 to the computation of edit distance between string $A =$ "ADGTF" and string $B =$ "AGCF" would result in the following $C$ matrix:

```
        A   D   G   T   F
    --------------------------
   | 0  1   2   3   4   5
   |
  A| 1  0   1   2   3   4
   |
  G| 2  1   1   1   2   3
   |
  C| 3  2   2   2   2   3
   |
  F| 4  3   3   3   3   2 <-- output.
```

Here, the edit distance $d(A, B)$ is $C[4, 5]$, i.e., 2.

The basic version of dynamic programming method was later extended to solve the *k-differences string-matching* problem. This is a similar but different problem: finding all similar occurrences of a pattern string in a longer string. Yet it turns out that the algorithm to tackle this problem needs only minor changes from the original algorithm described above. It was introduced in [Sellers79]. See Galil and Park [Galil90].

Since our new algorithm is directly based on this algorithm, we present it bellow.

Suppose we have a pattern string $A$ and a text string $T$ of lengths $m$ and $n$ respectively, and an integer $k$ ( $k \leq m \leq n$ ). Define $D[i, j]$ ( $0 \leq i \leq m$, $0 \leq j \leq n$ ) to be the minimum edit distance from $A_1^i$ to any substring of $T$ ending at $T_j$.

Similar to the case of algorithm 0, one can observe that for $i \leq m$ and $j \leq n$, the edit distance between $A_1^i$ and $T_h^j$ for some $h$ ($1 \leq h \leq j$) is either

(i) $d(A_1^{i-1}, T_h^{j-1}) + d(A_i, T_j)$, or
(ii) $d(A_1^i, T_h^{j-1}) + 1$ ( i.e., insert $T_j$ ), or
(iii) $d(A_1^{i-1}, T_h^j) + 1$ ( i.e., delete $A_i$ ).

Thus,

$$D[i,j] = \\ min(D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+c_{i,j}) \quad (2)$$

where $c_{i,j}$ denotes $d(A_i, T_j)$.

Initially, $D[i,0] = i$ for $0 \leq i \leq m$ because $A_1^i$ differs from the empty text by $i$ insertion steps. $D[0,j] = 0$ for $0 \leq j \leq n$ because the empty pattern occurs any where in $T$. $D[m,j] \leq k$ if and only if $A$ occurs at position $T_j$ with an edit distance at most $k$.

The following algorithm finds all substrings in $T$ with edit distances to $A$ less than or equal to $k$.

*Algorithm* 1:

```
Initialization:
    for $0 \leq i \leq m$ do $D[i,0] \leftarrow i$
    for $0 \leq j \leq n$ do $D[0,j] \leftarrow 0$
Main loop:
    for $i = 1$ to $m$ do
      for $j = 1$ to $n$ do
          $x \leftarrow D[i-1,j]+1$
          $y \leftarrow D[i,j-1]+1$
          if $A_i = T_j$ then
             $z \leftarrow D[i-1,j-1]$
          else
             $z \leftarrow D[i-1,j-1]+1$
          $D[i,j] \leftarrow \min(x,y,z)$
```

The last row of $D$ matrix, $D[m,j]$'s $(1 \leq j \leq n)$, is the output of this algorithm. More specifically, if a $D[m,j]$ is less or equal to $k$, then, there's an approximate occurrence of pattern $A$ in $T$ ending at position $j$, with an edit distance less than or equal to $k$.

Here, again, as in the case of original dynamic programming algorithm, the whole computation consists of filling out a matrix $D[0..m, 0..n]$, using the above formula iteratively. The difference between the two is that algorithm 1 initializes row zero with all 0's whereas algorithm 0 does not.

*Example 2.3.* Applying algorithm 1 to solve the *k-differences string-matching* problem for pattern string $A$ = "CCCF" and string $T$ = "ADGTFCF" would result in the following $D$ matrix (assume $k = 2$):

$$\begin{array}{c|cccccccc}
 & & A & D & G & T & F & C & F \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
C & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
C & 2 & 2 & 2 & 1 & 2 & 2 & 1 & 1 \\
C & 3 & 3 & 3 & 2 & 2 & 3 & 2 & 2 \\
F & 4 & 4 & 4 & 3 & 3 & 2 & 3 & 2
\end{array}$$

Here, the last row is the output. Since in that row only $D[4, 5]$ and $D[4, 7]$ are less or equal to 2, we know that there are approximate occurrences of pattern $A$ which ends at $T_5$ and $T_7$.

Obviously, these dynamic programming algorithms compute $C[i, j]'s$ or $D[i, j]'s$ row by row, and have time complexity of $O(m \times n)$.

Since the invention of these algorithms, there has been considerable progress on improving their computation efficiencies, resulting in many parallel and serial algorithms. We describe several algorithms here.

For algorithm 1, while the $D[i,j]'s$ cannot be computed row by row or column by column in parallel, they can be computed on a line parallel to anti-diagonal (i.e., all $D[i,j]'s$ with $i+j$ equal to some constant $d$ with $1 \leq d \leq m+n$) simultaneously. Let *D-antidiagonal(d)* denote those $D[i,j]'s$ such that $|j+i| = d$. Then, the matrix $D$ can be filled from the upper-left corner to the lower-right corner by *D-antidiagonals* with $d = 1, 2, \ldots, m+n$.

Guibas, Kung, and Thompson [Guibas79] gave a systolic array implementation of algorithm 0 for computing the edit distance between two strings. Lipton and Lopresti [Lipton85] described a VLSI systolic array implementation of the same algorithm that uses $2n+1$ processing elements to compute the edit distance between two strings of length $n$.

As described in [Lipton85], the $2n+1$ processors are arranged as a linear array. The characters from the two strings to be compared are fed into this processor array from left and right respectively. The two input streams are interleaved with the data values from column zero and row zero of matrix $C$, respectively. For example, if the two strings are "abd" and "dbb", their interleaved version would be "a1b2d3" and "d1b2b3", respectively. As is characteristic of systolic array architectures, the data is pumped through rhythmically according to a global synchronizing clock. In each processor, there is a state variable, which is initialized to 0. When two non-null characters meet in a processor from opposite direction, a comparison is performed. Then, on the next clock tick, the characters shift out and the data values following them shift in. This processor now computes the new value for the state variable, according to the previous value of the state variable (serving as the upper-left cell in matrix $C$), the result of comparison, and the two values just shifted in (serving as the left and upper cells), using the same rule as in *algorithm* 0. This new state variable then is used to update the interleaving values (both left-bounded and right-bounded). In this two-way shifting and computation process, $D$-cells are effectively computed along *D-antidiagonals.* The last value which is shifted out will be the lower-right-most $D$–cell, hence the answer.

The running time of their algorithm is the time from when a string is fed in to the time that string is completely out at the other end. Since the interleaved string is $2n$ long, the run time is $4 * n * s$, where $s$ is the maximum time need by a processor to perform its task on each clock tick.

*Example 2.4.*
The following figure depicts the computation of the edit distance between string “abd” and string “dbb”.

```
         0       0       0       0       0       0       0
       ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
3d2b1->| a |-->|   |-->|   |-->|   |-->|   |-->|   |-->|   |-->
    <--|   |<--|   |<--|   |<--|   |<--|   |<--|   |<--| d |<--1b2b3
       `___'   `___'   `___'   `___'   `___'   `___'   `___'


         1       0       0       0       0       0       1
       ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
3d2b-->| 1 |-->| a |-->|   |-->|   |-->|   |-->|   |-->|   |-->
    <--|   |<--|   |<--|   |<--|   |<--|   |<--| d |<--| 1 |<--b2b3
       `___'   `___'   `___'   `___'   `___'   `___'   `___'


         1       1       0       0       0       1       1
       ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
 3d2-->| b |-->| 1 |-->| a |-->|   |-->|   |-->|   |-->|   |-->
    <--|   |<--|   |<--|   |<--|   |<--| d |<--| 1 |<--| b |<--2b3
       `___'   `___'   `___'   `___'   `___'   `___'   `___'


         2       1       1      0(1)     1       1       2
       ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
  3d-->| 2 |-->| b |-->| 1 |-->| a |-->|   |-->|   |-->|   |-->
    <--|   |<--|   |<--|   |<--| d |<--| 1 |<--| b |<--| 2 |<--b3
       `___'   `___'   `___'   `___'   `___'   `___'   `___'
```

```
      2       2      1(1)     1      1(1)     2       2
    ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
3-->| d |-->| 2 |-->| b |-->| 1 |-->| a |-->|   |-->|   |-->
 <--|   |<--|   |<--| d |<--| 1 |<--| b |<--| 2 |<--| b |<--3
    `---'   `---'   `---'   `---'   `---'   `---'   `---'


      3      2(0)     2      1(0)     2      2(1)     3
    ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
 -->| 3 |-->| d |-->| 2 |-->| b |-->| 2 |-->| a |-->|   |-->
 <--|   |<--| d |<--| 2 |<--| b |<--| 2 |<--| b |<--| 3 |<--
    `---'   `---'   `---'   `---'   `---'   `---'   `---'


              2      2(1)     1      2(0)     3
    ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
 -->|   |-->| 2 |-->| d |-->| 1 |-->| b |-->| 3 |-->| a |-->
 <--| d |<--| 2 |<--| b |<--| 1 |<--| b |<--| 3 |<--|   |<--
    `---'   `---'   `---'   `---'   `---'   `---'   `---'


                      2      1(1)     2
    ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
 -->|   |-->|   |-->| 2 |-->| d |-->| 2 |-->| b |-->| 3 |-->a
d<--| 2 |<--| b |<--| 2 |<--| b |<--| 2 |<--|   |<--|   |<--
    `---'   `---'   `---'   `---'   `---'   `---'   `---'


                               2
     ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
  -->|   |-->|   |-->|   |-->| 2 |-->| d |-->| 2 |-->| b |-->3a
d2<--| b |<--| 2 |<--| b |<--| 2 |<--|   |<--|   |<--|   |<--
     `---'   `---'   `---'   `---'   `---'   `---'   `---'
```

```
        ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
     -->|   |-->|   |-->|   |-->|   |-->| 2 |-->| d |-->| 2 |-->b3a
  d2b<--| 2 |<--| b |<--| 2 |<--|   |<--|   |<--|   |<--|   |<--
        `---'   `---'   `---'   `---'   `---'   `---'   `---'


        ,---.   ,---.   ,---.   ,---.   ,---.   ,---.   ,---.
     -->|   |-->|   |-->|   |-->|   |-->|   |-->| 2 |-->| d |-->2b3a
 d2b2<--| b |<--| 2 |<--|   |<--|   |<--|   |<--|   |<--|   |<--
        `---'   `---'   `---'   `---'   `---'   `---'   `---'
                  ^
                  |
                  `-- result
```

In this figure, the boxes represent processing elements(PE). A value above a PE which is enclosed in a parenthesis represents the result of a comparison (0=success). Values above PE's but not in parentheses represent state variables.

For the $k$-differences string matching problem, Landau [Landau89] gave an interesting parallel algorithm which runs in $O(log(m)+k)$ using $n$ processors. Their method is a parallelization of a serial method given in [Ukkonen83] which computes the $D[i,j]'s$ along main diagonals. To explain their method, we need first to look at another algorithm, a variation of *algorithm* 1, which was first invented by Ukkonen [Ukkonen83].

The following description is somewhat reminiscent of [Landau89]. Let $A$ and $T$ denote the pattern string and text string, respectively. Let *D-diagonal(d)* of the matrix $D$ consist of all $D[i,l]$'s such that $l-i=d$. Let $e$ denote the number of differences, and let $L_{d,e}$ denote the largest row $i$ such that $D[i,l]=e$ and $D[i,l]$ is on diagonal $d$. By definition, $e$ is the minimum edit distance between $A_1^{L_{d,e}}$ and any substring of $T$ which ends at $T_{L_{d,e}}+d$. The definition of $L_{d,e}$ also implies that $A_{L_{d,e}+1} \neq T_{L_{d,e}+d+1}$ (for $L_{d,e}<m$).

*Example 2.5.* Continuing from example 2.3, we show the $L_{d,e}$ values for diagonal 3. They are: $L_{3,0} = 0$, $L_{3,1} = 1$, $L_{3,2} = 4$.

```
       A    D    G    T    F    C    F
  ,------------------------------------
  |              0
  |
 C|                   1
  |
 C|                        2
  |
 C|                             2
  |
 F|                                  2
```

For each diagonal $d$, we are interested in finding out the values of $L_{d,e}$'s, for $e = 0, \ldots, k$. If one of such $L'_{d,e}s$ equals $m$, then, there is an occurrence of pattern $A$ in $T$ with at most $k$ differences that ends at $T_{d+m}$.

In computing $L_{d,e}$'s, we use induction on $e$. For a given $d$ and $e$, suppose we've computed $L_{y,x}$, for all $x < e$ and all diagonals $y$. Suppose $L_{d,e}$ should get value $i$, i.e., $i$ is the largest row such that $D[i, l] = e$, and $D[i, l]$ is on diagonal $d$. By *algorithm* 1, there are two possibilities for $D[i, l]$ in getting its value $e$:

(i) $D[i-1, l-1]$ (predecessor of $D[i, l]$ on diagonal $d$) is $e - 1$ and $A_i \neq T_l$. Or, either $D[i, l-1]$ or $D[i-1, l]$ has a value of $e - 1$.

(ii) $D_{i-1}^{l-1}$ equals $e$ and $A_i = T_l$.

This implies that we may begin from $D[i, l]$ and go upward along diagonal $d$ by possibility (ii) until we get a possibility (i). Putting it in a reverse way, we can compute $L_{d,e}$ by first locating the row of diagonal $d$ where possibility (i) occurs, and then increasing the row number as long as possibility (ii) occurs.

These observations leads to the following algorithm:

*Algorithm* 2:

```
Initialization:
    for $0 \leq d \leq n$ do $L_{d,e} \leftarrow -1$
    for $-(k+1) \leq d \leq -1$ do $L_{d,|d|-1} \leftarrow |\, d \,| -1$, $L_{d,|d|-2} \leftarrow |\, d \,| -2$
    for $-1 \leq e \leq k$ do $L_{n+1,e} \leftarrow -1$
Main loop:
    for $e = 0$ to $k$ do
      for $d = -e$ to $n$ do
    1       $row \leftarrow \max[\ (L_{d,e-1}+1), (L_{d-1,e-1}), (L_{d+1,e-1}+1)\ ]$
            $row \leftarrow \min(row, m)$
    2       while $row < m$ and $row + d < n$ and $A_{row+1} = T_{row+1+d}$ do
                $row \leftarrow row + 1$
    3       $L_{d,e} \leftarrow row$
    4       if $L_{d,e} = m$ then
                print "There is an occurrence ending at $T_{d+m}$"
```

As can be seen, this algorithm computes $L_{d,e}$'s along $n + k + 1$ diagonals. Furthermore, since in the inner loop, the variable $row$ can take at most $m$ different values, this algorithm has a time complexity of $O(mn)$ (although in practice, this algorithm might be faster than the basic algorithm, due to the fact it never computes values for $D$ cells which have cost greater than $k$).

Now we can explain what Landau and Vishkin did in [Landau89].

Basically, their algorithm is a parallelization of algorithm 2. Observe that in the inner loop of algorithm 2, statement 2 is a while loop which carries the task of finding $L_{d,e}$. The first idea in Landau and Vishkin's algorithm is that if we can find $L_{d,e}$ in $O(1)$ time instead of using this while loop, the whole time complexity would be reduced to $O(nk)$ (in a serial sense).

The trick is utilizing a suffix tree which is defined as follows.

*Definition:*

Let $S = s_1, \ldots, s_p$ be a string where $s_p =$ "\$" and "\$" "\$" does not appear elsewhere in $S$. The suffix tree of $S$ is defined as:

(1) Each suffix of $S$, $S_i^p$, defines a leaf node in the tree.

(2) The empty string defines the root of the suffix tree.

(3) For any pair of leaves, $S_i^p$ and $S_j^p$, their longest common prefix (which could be empty string, i.e., the root) defines their immediate parent node in the tree. More generally, for any pair of nodes, their longest common prefix defines their immediate parent node in the suffix tree.

Each node $v = S_i^j$ can be stored as: START($v$) $= i - 1$ and LENGTH($v$) $= j - i + 1$, recording node $v$'s start position and length. Also, each node should keep pointers to its immediate parents.

*Example 2.6.* the suffix tree for string abab\$ is:

```
                    (root)

                  /   |   \
                 /    |    \
                /     |     \
               /      |      \
              /       |       \
             /        |        \
            /         |         \
        (ab)         (b)         ($)

        /  \         /  \
       /    \       /    \
  (abab$) (ab$)  (bab$) (b$)
```

It is stored as: START($ab$) = 2, LENGTH($ab$) = 2, START($b$) = 3, LENGTH($b$) = 1, START(\$) = 4, LENGTH(\$) = 1, START($abab\$$) = 0, LENGTH($abab\$$) = 5, START($ab\$$) = 2, LENGTH($ab\$$) = 3, START($bab\$$) = 1, LENGTH($bab\$$) = 4, START($b\$$) = 3, LENGTH($b\$$) = 2.

The parallel algorithm developed in [Landau89] proceeds in two steps.

In the first step, the text string $T$ and pattern string $A$ are concatenated into one string $T_1, \ldots, T_n, A_1, \ldots, A_m\$$ where \$ is not an element of either $A$ or $T$. Then, the suffix tree of this concatenated string can be computed using the algorithm given in [Apostolico88] and [Landau87] in $O(\log n)$ time using $n$ processors.

In the second step, $n+k+1$ processors are employed, each dealing with a $D$-diagonal $d$, $-k \leq d \leq n$. The details are given as algorithm 3.

*Algorithm* 3:

```
Initialization: as in algorithm 2.
    for $e = 0$ to $k$ do
      for $d = -e$ to $n$ parallel-do
    1       $row \leftarrow \max[\ (L_{d,e-1} + 1), (L_{d-1,e-1}), (L_{d+1,e-1} + 1)\ ]$
            $row \leftarrow \min(row, m)$
    2       $L_{d,e} \leftarrow row +$ LENGTH$(LCA_{row,d})$
    3       if $L_{d,e} = m$ then
               print "There is an occurrence ending at $T_{d+m}$"
```

where, $LCA_{row,d}$ is the lowest common ancestor of the leaf nodes $T_{row+d+1} \ldots \$$ , and $A_{row+1}, \ldots \$$, in the suffix tree.

The differences in algorithm 3 from algorithm 2 are: (a)the inner for-loop is now done in parallel; (b)statement 2 in algorithm 2, i.e., the while statement , is replaced by a single assignment statement which finds $L_{d,e}$ in $O(1)$ time.

After statement 1, we know that on diagonal $d$, on row $row$, the edit distance (i.e., $D[row, row + d]$) is $e$. Then, we compute $L_{d,e}$, the largest row on diagonal $d$ such that $D[row, row + d]$ is still $e$. In algorithm 2, we achieve this by continuously increasing the value of $row$ as long as $T_{row+d+1} = A_{row+1}$. It is not difficult to see that the number of times we can increase the value of $row$ is exactly LENGTH$(LCA_{row,d})$. That is why the while-loop in algorithm 2 can be reduced to statement 2 in algorithm 3.

*Example 2.7.* Continuing from example 2.3, in computation of $L_{3,2}$, the values of $L_{2,1}$,$L_{3,1}$,$L_{4,1}$ are needed. Since their values are 1, 1 and 2, respectively, by algorithm 3, we initialize *row* to max[ $L_{2,1}$, $L_{3,1} + 1$, $L_{4,1} + 1$ ] = 3 (see the following figure). Next, we look at LENGTH($LCA_{row,d}$), i.e,LENGTH($LCA_{3,3}$). Since $LCA_{3,3}$ is the longest common prefix of the two leaves $T_7 \ldots \$$ (=FCGCCF\$) and $A_4, \ldots \$$ (=F\$), we know from the suffix tree that LENGTH($LCA_{3,3}$)=LENGTH(F)=1. Thus, $L_{3,2}$ = row + LENGTH(F) = 3 + 1 = 4.

```
        A    D    G    T    F    C    F
  ,-------------------------------------
  | 0        0    0    0

  |
 C| 1             1    1    1
  |
 C| 2                  2    2    1
  |
 C| 3                       3    2    2  <---initially,row = 3
  |
 F| 4                            3    2
```

Since $L_{3,2} = 4 = m$ and $e = 2 \leq k$, this algorithm will report an occurrence of pattern $A$ which ends at $T_{d+m} = T_7$.

In algorithm 3, the time complexity is $O(k)$. Since the time needed to construct the suffix tree is $O(\log n)$[Apostolico88], the total time is $O(k + \log n)$[Landau89].

Finally, using the divide-and-conquer method, one can split the original problem into $\lceil n/m \rceil$ smaller problems each of size $O(m)$, and solve them in parallel using the above algorithm and $\lceil n/m \rceil$ groups of processors, thus achieving a time complexity of $O(k + \log m)$.

For the sake of completeness, let's have a look at another interesting development, a serial algorithm given in [Galil90] by Galil and Park , which has a time complexity of $O(kn + m^2)$.

The following is a description of their method.

First, note that on each diagonal $d$, $D[i,j]'s$ are non-decreasing (see lemma 1, subsection 3.4). This feature of $D$ matrix suggests a more compact way of storing the information in $D$. That is, on each *diagonal* $d$, we store only those positions where the value actually increases. In this thesis, we will refer to those positions in $D$ after which (on the same diagonal) the value increases from $e-1$ to $e$ as *jump points* of $e-1$.

For a difference $e$, let $E[e,d]$ be the largest column $j$ such that on diagonal $d$, $D[j-d,j]$ equals $e$. In other words, $D[E[e,d]-d, E[e,d]]$ is the jump point on matrix $D$ for difference $e$. Then, if $E[e,d]-d = m$ for some $e \leq k$, we know that there's an approximate occurrence of pattern $A$ which ends at column $m+d$.

Now, the original computation on a $m \times n$ matrix $D$ has been transformed into computation on an order $k \times n$ matrix $E$. This computation of matrix $E$ can be done by the following algorithm (if we think in terms of $D$ matrix, this is just another representation of algorithm 2):

*Algorithm* 4: (This isn't their new algorithm; Algorithm 5 is)

```
Initialization:
    for $0 \leq d \leq n-m+k+1$ do $E[-1,d] \leftarrow d-1$
    for $-(k+1) \leq d \leq -1$ do
            $E[\mid d \mid -1, d] \leftarrow -1$
            $E[\mid d \mid -2, d] \leftarrow -\infty$
Main loop:
    for $c = 0$ to $n-m+k$ do
    1   for $e = 0$ to $k$ do
            $d \leftarrow c - e$
    2       $col \leftarrow \max(E[e-1,d-1]+1,\ E[e-1,d]+1,\ E[e-1,d+1])$
    3       while $col < n$ and $col - d < m$ and $A_{col+1-d} = T_{col+1}$ do
                $col \leftarrow col + 1$
    4       $E[e,d] \leftarrow \min(col, m+d, n)$
            if $E[e,d] = m+d$ then
```

```
print "There is an occurrence ending at $T_{d+m}$"
```

Note that the row index of matrix $E$ ranges from -1 to $k$, whereas the column index extends from $-(k+1)$ to $n-m+k+1$.

By nature, algorithm 4 is equivalent to algorithm 2 in that both determine the jump point for a difference $e$ by first determining a cell achieving $e$ by looking at $e-1$ cells on this and adjacent $D$-diagonals, and then proceed along this $D$-diagonal as long as the corresponding elements from $A$ and $T$ matches. However, a big difference exists: in algorithm 4, we compute $E$ cells (the jump points on $D$-diagonals) by antidiagonals on $E$; If we translate $E$ cells into corresponding $D$ cells, we'll notice that the order of computation is quite different.

Following the convention in the original paper, we call antidiagonals on $E$ as $E-diagonals\ c$, where for each cell $E[i,j]$ on $E$, $i+j = c$.

The $E$ matrix contains three types of columns:

(1) For $-k \leq d \leq -1$, only $d+k+1$ cells (i.e.,$E[e,d]$, $\mid d \mid \leq e \leq k$) need to be considered, since $D-diagonal\ d$ on matrix $D$ begins with value $d$.

(2) For $0 \leq d \leq n-m$, $k+1$ cells (i.e., $E[e,d]$ $0 \leq e \leq k$) need to be computed.

(3) For $n-m+1 \leq d \leq n-m+k$, $(n-m+k)-d+1$ cells (i.e., $E[e,d]$ $0 \leq e \leq (n-m+k)-d$) need to be computed, since that although these corresponding $D-diagonals$ lead to no results, their values may nevertheless affect those $D-diagonals$ below them on matrix $D$.

In the initialization stage of algorithm 4, special values, $-1$, and $-\infty$ are used. Since on $D$ matrix, the situation $e=0$ begins at column $d$ on $D-diagonal\ d$, it is proper to initialize $E[-1,d]$ to $d-1$ (for $d \geq 0$). Since for $-(k+1) \leq d \leq -1$, the $D-diagonal$ $d$ begin with value $\mid d \mid$ at column 0 on matrix $D$, we initialize $E[\mid d \mid -1, d]$ to $-1$ (that is, value $-1$ on $D-diagonal\ d$ ends at column $-1$—an imaginary column, of course). For reason of dependencies among cells on $E$, we also initialize $E[\mid d \mid -2, d]$ to -$\infty$, for $-(k+1) \leq d \leq -1$ .

*Example 2.8.* Let pattern string $A$ = "CCCF", text string $T$ = "ADGTFCF"and $k = 2$ again. The initial $E$ matrix would be:

| e \ d | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| -1 | | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | | $-\infty$ | -1 | | | | | | | |
| 1 | $-\infty$ | -1 | | | | | | | | |
| 2 | -1 | | | | | | | | | |

From algorithm 4, we see that $E[i-1, j-1]$, $E[i-1, j]$, and $E[i-1, j+1]$ (Statement 2) are used to compute $E[i, j]$. That is, every cell on $E$ is dependent on three cells above it in the previous row.

*Lemma 2.1.* In the computation of $E-diagonal\ c$ , the elements of $T$ which are compared with the pattern are at most $T_{c+1}, \ldots, T_{min(c+m,n)}$.

Proof. In the computation of $E[e, c-e]$ $(0 \le e \le k)$, consider the first element of $T$ to be compared. Since the computation of $E[e, c-e]$ begins with the comparison of $T_{c-e+1}$, and there is at least one cell on $D - diagonal\ c-e$ for each $e'$, $0 \le e' < e$, the first position of $T$ to be compared in the computation of $E[e, c-e]$ has to be greater than or equal to $c+1$. Now consider the last element of $T$ to be compared in the computation of $E[e, c-e]$. Since the last column position is $m+c-e$ for $D - diagonal\ c-e$, the last position of $T$ elements compared against the pattern has to be less than or equal to $m+c-e$, $0 \le e \le k$.

*Lemma 2.2.* For the computation of a $E - diagonal\ c$,

(1) The position of $T$'s elements which are compared against the pattern is nondecreasing.

(2) Any position in $T$ can at most repeat $k$ times.

Proof. For $0 \leq e \leq k$, $E[e, c-e]$ is the last position in $T$ with respect to that $e$, and $E[e, c-e]+1$ is the last position of $T$ involved in comparison. We need to show that the position of the first element compared in computation of $E[e+1, c-(e+1)]$ will be at least $E[e, c-e]+1$.

From statement 2 in algorithm 4, we know that at the beginning of computation of $E[e+1, c-e-1]$, $col \geq E[e, c-e]$ (substitute $e+1$ and $c-e-1$ for $e$ and $d$ in statement 2, respectively), then we have $col+1 \geq E[e, c-e]+1$, that is , the first comparison for $e+1$ is at a position greater than or equal to that of the last comparison for $e$. Hence lemma 2.2(1).

Since repetitions are only possible at the beginning of computation for each $e$, $1 \leq e \leq k$, there can be at most $k$ repetitions.

By lemma 2.1 and 2.2, the loop on $e$ in algorithm 4 needs time $O(m)$, thus the total time complexity of algorithm 4 is $O(mn)$.

Based on algorithm 4, [Galil90] gives an improved (serial) algorithm which solves the problem in time $O(kn)$.

As in the situation of algorithm 3, the idea is: instead of comparing the $T$ elements against $A$ elements one at a time, we try to jump as far as we can down the $D$ – diagonal using some kind of mechanism (algorithm 3 does this perfectly by using a suffix tree, but has the overhead of having to build the suffix tree). In this case, a *prefix* table is used. It isn't as good as a suffix tree, but is easier to build.

Let $Prefix[i, j]$, $1 \leq i < j \leq m$, be the length of the longest common prefix of $A_i^m$ and $A_j^m$. This is an upper-triangular matrix.

*Example 2.9.* Continuing from example 2.8, the prefix matrix of pattern string $A =$ "CCCF" is:

| i \ j | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |  | 2 | 1 | 0 |
| 2 |  |  | 1 | 0 |
| 3 |  |  |  | 0 |
| 4 |  |  |  |  |

*Reference triples* are also used in their algorithm, in order to utilize the *prefix* matrix.

A *reference triple* $(u, v, w)$ is an array with 3 elements, the first of which specifies a start position, the second an ending position, and the third a $D-$ *diagonal* $w$ such that $T_u^v$ matches $A_{u-w}^{v-w}$, and $T_{v+1} \neq A_{v+1-w}$. The triple is a *null* triple if $u > v$.

A *reference position*, with respect to a reference triple $(u, v, w)$ and a position $t$ in $T$ $(u \leq t \leq v)$, is the position $t - w$ in pattern $A$.

Let's consider again the computation of $E-$ *diagonal* $c$, that is, $E[e, c-e], 0 \leq e \leq k$. Let $T_{q+1}$ be the rightmost element in $T$ which was compared before we enter iteration $c$. Assume that we have $k+1$ reference triples, $(u_0, v_0, w_0), (u_1, v_1, w_1), \ldots, (u_k, v_k, w_k)$, constructed in earlier loops, and they form a partition of the interval $[c, q]$.

The following algorithm utilizes the prefix matrix and the reference triples to compute the same information as does algorithm 4.

*Algorithm 5.*

```
Initializations as in algorithm 4.
Main loop:
    for $c = 0$ to $n - m + k$ do
```

```
     $r \leftarrow 0$
1    for $e = 0$ to $k$ do
         $d \leftarrow c - e$
2        $col \leftarrow \max(E[e-1, d-1]+1,\ E[e-1, d]+1,\ E[e-1, d+1])$
         $s_e \leftarrow col + 1$
         $found \leftarrow$ false
3        while not $found$ do
4            if within($col+1, k, r$) then
                     $f \leftarrow v_r - col$
5                    $g \leftarrow Prefix[col+1-d, col+1-w_r]$
6                    if $f = g$ then
                         $col \leftarrow col + f$
                     else
                         $col \leftarrow col$+min($f, g$)
                         $found \leftarrow$ true
7            else
                     if $col < n$ and $col - d < m$ and $T_{col+1} = A_{col+1-d}$ then
                         $col \leftarrow col + 1$
                     else
                         $found \leftarrow$ true
8        $E[e, d] \leftarrow \min(col, m+d, n)$
         // update reference triple $(u_e, v_e, w_e)$
         if $v_e \geq E[e, d]$ then
             if $e = 0$ then $u_e \leftarrow c + 1$
             else $u_e \leftarrow \max(u_e, v_{e-1} + 1)$
         else
             $v_e \leftarrow E[e, d]$
             $w_e \leftarrow d$
             if $e = 0$ then $u_e \leftarrow c + 1$
             else $u_e \leftarrow \max(s_e, v_{e-1} + 1)$


function $within(t, k, r)$
     while $r \leq k$ and $t > v_r$ do
             $r \leftarrow r + 1$
     if $r > k$ then return(false)
     else
```

```
if $t \geq u_r$ then return(true)
else return(false)
```

At iteration $c$ of the main loop, as before, our focus is at variable *col*, which is first taken as the maximum of three values obtained previously, and then starting from that position, going down the $D$ diagonal, we look for the first occurrence of the situation that $T$ element and $A$ element mismatch. Instead of using a while loop to go down the $D$ diagonal $d$ as long as the corresponding $T$ and $A$ elements matches, we first check whether the current position $t$ ($= col + 1$) is within the range of one of the reference triples. If it's not, we have no other way than to do the actual comparison(s) between $T$ and $A$ elements. If, however, it is within a reference triple $(u_r, v_r, w_r)$, i.e., $u_r \leq t \leq v_r$, then, we can use $prefix$ table to "jump" down the diagonal $d$ ($d = c - e$).

Now, the current pattern position is $t - d$. The reference position corresponding to $t$ is $t - w_r$. By the definition of a reference triple, we can infer that $A_{t-w_r}^{v_r-w_r} = T_t^{v_r}$. Let $g$ be $Prefix[t-d, t-w_r]$. We want to find the largest $x$ such that $A_{t-d}^{x-d} = T_t^x$.

Since $A_{t-w_r}^{t-w_r+g} = A_{t-d}^{t-d+g}$ (definition of $Prefix[]$), we must have $A_{t-d}^{min(v_r,t+g)-d} = T_t^{min(v_r,t+g)}$. Thus, we can safely advance *col* to $\min(v_r, t+g)$. Next, if $v_r < t+g$, we know $A_{v_r+1-d} \neq T_{v_r+1}$, since we have $A_{v_r-w_r+1} \neq T_{v_r+1}$ (by definition of a reference triple) and $A_{v_r-d+1} = A_{v_r-w_r+1}$. If $v_r > t+g$, we have $A_{t+g+1-d} \neq T_{t+g+1}$, since $A_{t+g+1-w_r} = T_{t+g+1}$ but $A_{t+g+1-d} \neq A_{t+g+1-w_r}$.

Thus, if $v_r \neq t+g$, we know for sure that $E[e,d] = min(v_r, t+g)$. In case of $v_r = t+g$, we don't know whether $A_{v_r+1-d} = T_{v_r+1}$, we have to continue from $v_r + 1$ to do further actual comparison or try another reference triple whenever *col* falls into the range of a reference triple.

At the end of the calculation of each $E[e,d]$, we need to update $(u_e, v_e, w_e)$. Let $s_e$ be the first position of $T$ considered in computation of $E[e,d]$. Then, by the way $E[e,d]$ is computed, we must have $A_{s_e-d}^{E[e,d]-d} = T_{s_e}^{E[e,d]}$. Obviously, $(s_e, E[e,d], d)$ is a potential candidate for the replacement of $(u_e, v_e, w_e)$, $0 \leq e \leq k$. But we can not immediately do the replacement, for there may be multiple holes between $E[e,d]$ and $s_{e+1}$ (i.e., when $s_{e+1} > E[e,d] + 2$). We solve this problem by choosing the triple which has the larger end position, e.g., if $E[e,d] > v_e$, we replace old triple, otherwise, we don't replace. Whether we replace old triple or not, we need to adjust $u_e$ so that this triple does not overlap with the previous one.

Initially, all reference triples are $(0, 0, 0)$.

*Example 2.10.* Let's continue from example 2.8 and 2.9 ($A$ = "CCCF",$T$ = "ADGTFCF" and $k = 2$). The figures bellow show the $E$ matrix and the reference triple at the end of each iteration $c$, $0 \leq c \leq 5$.

```
initial state:

E:
,-------------------------------------------------------------
| e \ d |     -3       -2       -1      0     1    2    3    4    5    6
|-------------------------------------------------------------
|  -1   |                       −∞     -1     0    1    2    3    4    5
|       |
|   0   |              −∞       -1
|       |
|   1   |     −∞       -1
|       |
|   2   |     -1

    reference triple:    (0 0 0)    (0 0 0)    (0 0 0)

  c = 0:
```

```
E:

,----------------------------------------------------------------
| e \ d |    -3     -2     -1     0    1    2    3    4    5    6
|----------------------------------------------------------------
|  -1   |                  −∞    -1    0    1    2    3    4    5
|       |
|   0   |           −∞     -1     0
|       |
|   1   |    −∞     -1      0
|       |
|   2   |    -1      0

   reference triple:    (1 0 0)   (1 0 0)   (1 0 0)


 c = 1:

 E:

,----------------------------------------------------------------
| e \ d |    -3     -2     -1     0    1    2    3    4    5    6
|----------------------------------------------------------------
|  -1   |                  −∞    -1    0    1    2    3    4    5
|       |
|   0   |           −∞     -1     0    1
|       |
|   1   |    −∞     -1      0     1
|       |
|   2   |    -1      0      1

   reference triple:    (2 1 1)   (2 2 0)   (3 1 -1)
```

c = 2:

E:

| e \ d | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| -1 | | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | | $-\infty$ | -1 | 0 | 1 | 2 | | | | |
| 1 | $-\infty$ | -1 | 0 | 1 | 3 | | | | | |
| 2 | -1 | 0 | 1 | 3 | | | | | | |

reference triple: (3 2 2) (3 3 1) (4 3 0)

c = 3:

E:

| e \ d | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| -1 | | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | | | |
| 1 | $-\infty$ | -1 | 0 | 1 | 3 | 3 | | | | |
| 2 | -1 | 0 | 1 | 3 | 5 | | | | | |

reference triple: (4 3 3) (4 3 1) (5 5 1) c = 4:

E:

| e \ d | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| -1 | | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | 4 | | |
| 1 | $-\infty$ | -1 | 0 | 1 | 3 | 3 | 4 | | | |
| 2 | -1 | 0 | 1 | 3 | 5 | 4 | | | | |

reference triple: (5 4 4) (5 4 3) (5 5 1)

c = 5:

E:

| e \ d | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| -1 | | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | | $-\infty$ | -1 | 0 | 1 | 2 | 3 | 4 | 5 | |
| 1 | $-\infty$ | -1 | 0 | 1 | 3 | 3 | 4 | 6 | | |
| 2 | -1 | 0 | 1 | 3 | 5 | 4 | 7 | | | |

reference triple: (6 5 5) (6 6 4) (7 7 3)

In the last figure ($c = 5$), in the last row, note that for $e = 2$,$d = 1$, $E[e, d] = E[2, 1] = 5 = 4 + 1 = m + d$, since $e \leq k$, we know there is an approximate occurrence of pattern $A$ in $T$, with an edit distance less than or equal to $k$, and which ends at column 5. The same applies to $E[2, 3]$. That is, for $e = 2$, $d = 3$, $E[e, d] = 7 = 4 + 3 = m + d$, since $e \leq k$, we know there is an approximate occurrence of pattern $A$ in $T$, which ends at column 7.

*Complexity analysis.* For each iteration $c$, consider the number of repetitions of the while loop in this algorithm: it is the number of direct comparisons plus the number of lookups of the $Prefix$ table.

How many direct comparisons are needed at most? There're two cases in which a direct comparison is needed:

(1) If $t > q + 1$ (i.e., $T_t$ is a newly seen element);

(2) $t \leq q+1$ but $t$ falls into one of the holes between two adjacent reference triples.

For case one, by Lemma 2.1, the number of repetitions for a position $t$ can not exceed $k$ in iteration $c$. Also, at the next iteration, position $t$ will become in case (2). So, there can be at most $O(k * n)$ overall direct comparisons in case (1) in the whole computation.

For case two, since there can be at most $k$ holes in interval $[c, q + 1]$, at most $k$ direct comparisons at a hole can occur at iteration $c$. Thus, the overall number of such comparisons in the whole computation can be at most $O(k * n)$.

Now, how many $Prefix$ table lookups are needed at most? At iteration $c$, since a lookup of $Prefix$ table either increases $e$ or $r$, and $r$ can only take values less than $k + 1$, there can be at most $O(k)$ lookups during iteration on $E$ diagonal $c$. Hence, there can be at most $O(k * n)$ table lookups in the whole computation.

Conclusion: Algorithm 5 has time complexity $O(kn)$. It is so far the best serial algorithm in terms of worst-case complexity for solving the $k - difference$ problem.

## 3 A new systolic algorithm

### 3.1 Motivation of the new algorithm

The inherent parallelism in dynamic programming algorithms, and the rapid progresses in parallel processing technologies makes parallelization of these algorithms an obvious choice.

Although the time complexity of Landau's algorithm in [Landau89] is quite small: $O(log(m) + k)$, the algorithm uses $O(n)$ processors, which creates serious problem in some situations. For example, the text $T$ to be searched may well be residing on a hard disk. Then, how can you present the whole text "at once" to the $O(n)$ processor array? If you can't, if you have to feed the $n$ elements serially, then the actual time complexity will be something like $O(n)$. In addition to this "data path" problem, the usual assumption of $n \gg m$ may make the construction of a processor array of that size difficult.

So, in many cases, if we can reverse the orders of computation time and number of processors required in Landau's algorithm, e.g., if we use $m$ processors, and do the computation in $O(n)$ time, then, it will be more practical and economical: since we need $O(n)$ time in feeding the text to our computing device any way, the $O(n)$ time in computation is the fastest we can achieve in those cases.

This is the motivation of the new algorithm.

### 3.2 The new algorithm

By using the algorithm described in the following, we can solve the approximate string matching problem with at most k-differences, in time $s \times n$, using $m$ simple processing elements (called $PE$'s afterwards), where $s$ is very small time constant in which a few simple additions, comparisons, and inter$PE$ communications can be done.

The new algorithm is a parallelization of Algorithm 1. Processing element $PE_i$ is assigned to the computation of row $i$ of matrix $D$. On the $t^{th}$ time step, processor $PE_i$ computes $D[i, j]$ where $i + j = t + 1$. Thus, all elements on a secondary diagonal of $D$ are being computed at the same time. (See Figure 1.)

```
                              T
   .---------------------------------------------------.
   !       /                                           !
A  !      / ---->                                      !
   !     /                                             !
   `---------------------------------------------------'
      !
      `-> output D[m,t-m+1],  t=m, m+1, ... n+m-1
```

Figure 1. How the processors pass over the matrix D

The computation of $D[i,j]$ requires $D[i-1,j]$ and $D[i,j-1]$, which were computed on the previous time step, and $D[i-1,j-1]$ which was computed two time steps previously. Also required are $A_i$, which can be preloaded into $PE_i$ before the algorithm starts, and $T_j$, which was used by $PE_{i-1}$ on the previous time step. Thus, the required data for the computation of $D[i,j]$ comes from $PE_{i-1}$ and $PE_i$ on the previous two time steps.

The $m$ $PE$'s are connected into a linear array with one-way communication channels between successive $PE$'s. (See Figure 2.) Text character $T_t$ is fed into $PE_1$ on time step $t$. The output at time step $t$ is $D[m, t-m+1]$, which is the minimum edit distance from string $A$ to any substring of $T$ ending at $T_{t-m+1}$.

Interconnection topology (linear array):

$$\begin{tikzcd}
 & & {} \arrow[d] & \text{data stream } T \arrow[l, dash] \\
 & {} \arrow[r] & PE_1 \arrow[d] & \\
A \arrow[r] & {} \arrow[r] & PE_2 & \\
 & & \vdots \arrow[d] & \\
 & {} \arrow[r] & PE_m \arrow[d] & \\
 & & \text{output stream} &
\end{tikzcd}$$

Figure 2. Interconnection Topology (linear array)

These observations lead to the following algorithm:

*Algorithm 6* (a systolic algorithm):

Processors: $PE_i$, $i = 1, 2, \ldots, m$

Each $PE$ has integer memory locations $x, y, z, c, cost$, and character memory locations $a$ and $b$. These correspond to $D[i, j-1]$, $D[i-1, j]$, $D[i-1, j-1]$, $c_{i,j}$, $D[i, j]$, $A_i$, and $T_{t-i+1}$ respectively.

Each $PE$ has an NORTH input channel and a SOUTH output channel. For $i \neq 1$, memory locations $y$, $z$, and $b$ are buffer locations for the NORTH channel. They will contain the first, second, and third data "items" sent from the PE's north neighbor.

For $PE_1$, memory location $b$ is a buffer for the NORTH channel. It will contain the element $T_t$ sent from the $T$ input stream. For $t > n$, $b$ can contain an arbitrary character, since this character does not affect the output.

Initial configuration:

```
for $PE_i$, $i$ = 1, ... $m$ do
    $x$    ← $i$
    $cost$ ← $i$
    $a$    ← $A_i$
    $b$    ← $B^*$
    //Note: select $B^*$ to be a character that does not occur in $A$.

for $PE_1$ do
    $y$    ← 0
    $z$    ← 0
    //these values remain constant
```

We assume there is a system clock which generates a steady clock signal to each $PE$. A clock cycle is further divided into 2 subintervals, called subcycle 1 and 2 respectively. $PE$'s perform the following algorithm within a clock cycle.

Input: $T_t$, written to NORTH of $PE_1$ at subcycle 1, clock $t$.
Output: $D[m, t-m]$ at clock $t$, from SOUTH channel of $PE_m$.

```
int $x, y, z, c, cost$
char $a, b$
// $x, y, z, a, b$ contain $D[i, j-1]$, $D[i-1, j]$, $D[i-1, j-1]$, $A_i$, $T_j$

repeat
  in subcycle 1, do
    send(SOUTH, $cost$)        // goes to $y$ in the next $PE$
    send(SOUTH, $x$)           // not applicable when $i = m$
    send(SOUTH, $b$)           // not applicable when $i = m$
  in subcycle 2, do
    $x \leftarrow cost$                 // update "left" cell
    if $a = b$ then $c \leftarrow 0$ else $c \leftarrow 1$
    $cost \leftarrow min(x+1,\ y+1,\ z+c)$
```

The initialization using the special character $B^*$ is designed so that the values of the variables $x$, $y$, and $z$ in $PE_i$ remain unchanged when the time step $t \leq i$.

After the first data transfer between $PE$'s, the $y$ and $z$ values in $PE_i$ are $i-1$. Since $c = 1$ (for $B^*$ is a special character not in the pattern), the *cost* in $PE_i$ is chosen as the minimum of $x = i$, $y + 1 = i$, and $z + 1 = i$. Thus, $x$ retains the value $i$. This remains true for $PE_i$ as long as $t$ is less than $i$.

Using a special character $B^*$ may require that the maximum alphabet size be increased by one. An alternative is to start $PE_i$ on time step $i$.

After producing garbage at clock cycle 1 through $m$, this systolic algorithm will produce a useful result at each clock cycle until clock cycle $n + m$.

Clearly, the algorithm can be generalized to compute the edit distance where the cost of insertions, deletions, and substitutions are different from 1, including the case where these costs depend on the pattern or text characters.

## 3.3 Complexity analysis

Let $s$ be the time duration of one clock cycle. Obviously, the time complexity of this algorithm is $s \times (n + m - 1)$, and the number of $PE$'s used is $m$. Since the instructions executed within each clock cycle are fixed, the algorithm is real-time.

## 3.4 A further simplification of the processor complexity

Lipton and Lopresti [Lipton85] described a way to further simplify the processor complexity by using mod-4 arithmetic in the cost computations. This allows the integer memory locations of the $PE$'s to be 2-bit integers, thus greatly reduces the complexity of each $PE$.

Assuming the cost of an insertion and deletion to be one, and the cost of a substitution to be 2, their method relies on a lemma which states that the values of horizontal and vertical neighbors in the matrix $C$ (of *Algorithm* 0) *always* differ by 1 or -1.

Although the algorithm of this thesis is different, a similar approach can be applied.

Let $l$ be the cost of substitution. We consider two cases: $l = 1$, and $l = 2$ (the former is by far the most commonly studied case in the literature. The latter is used in Lipton and Lopresti's paper. One can prove that by putting the cost of substitution to be the same as the combined costs of an insertion and a deletion, the substitution can be essentially disregarded).

For $l = 1$, their lemma definitely does not hold here, for the difference between horizontal or vertical neighbors may zero. Even for $l = 2$, because of the difference in initialization, their lemma no longer holds , either.

However, we can prove a somewhat weaker but similar result(lemma 3.2 below). And to make things work, we need some other lemmas, too.

*Lemma 3.1* [Ukkonen85b, Galil90]: If the cost of a substitution is 1, then for every $i$ and $j$, $D[i,j] = D[i-1,j-1]$ or $D[i-1,j-1]+1$.

We defer the proof of lemma 3.1 until after the introduction of lemma 3.2.

Now we prove lemma 3.2.

*Lemma 3.2*: In matrix $D$, the values of horizontal and vertical neighbors differ by at most 1.

*Proof of Lemma 3.2*:

First, consider the case of $l = 1$.
Consider $D[i,j]$ and $D[i,j-1]$.
By the definition of $D[i,j]$, it cannot be more than $D[i,j-1]+1$.

Now suppose $D[i,j-1] - D[i,j] \geq 2$. Then, the value of $D[i,j]$ cannot come from the value of $D[i-1,j-1]$, because otherwise $D[i-1,j-1]$ would be less than $D[i,j-1]$ by more than one: contradictory to the definition of $D[i,j-1]$. So, the value of $D[i,j]$ must come from the value of $D[i-1,j]$, that is we must have $D[i,j] = D[i-1,j]+1$. This implies that $D[i-1,j]$ must be less than or equal to $D[i,j-1]$ by more than three, i.e., $D[i,j-1]-D[i-1,j] \geq 3$. Now let's look at the value of $D[i-1,j-1]$: dy definition, it must be greater than or equal to $D[i,j-1]-1$, i.e., $D[i-1,j-1]-D[i,j-1] \geq -1$. Combining the last two inequalities together, we have $D[i-1,j-1]-D[i-1,j] \geq 2$.

So, if we assume $D[i,j-1] - D[i,j] \geq 2$, we must have $D[i-1,j-1]-D[i-1,j] \geq 2$. Using induction, it follows that, in general, the inequality $D[p,j-1] - D[p,j] \geq 2$ (for $0 \geq p \geq i$) must be true.

But we know that, for $p = 0$, this is not true. Here is a contradiction which means that our original assumption is false. That is, $D[i,j]$ cannot be less than $D[i,j-1] - 1$.

So,Lemma 3.2 holds for horizontal neighbors.

Using the same kind of reasoning, we can prove that lemma 3.2 holds for vertical neighbors.

This completes the proof for case $l = 1$.

Second, consider the case of $l = 2$. We prove it by induction on $i + j$, the sum of the row and column indices.

For the basis, take $i+j$=2. In this case, $D[1,1]$ is either 0 or 1 and lemma 3.2 trivially holds.

For the induction hypothesis: assume lemma 3.2 is true for $i+j < p$.

Induction: consider the case $i + j = p$.

First consider vertical neighbors. If $| D[i,j] - D[i-1,j] | \geq 2$, then, since $D[i,j]$ is the minimum of three values which includes $D[i-1,j]+1$, it can only be that $D[i-1,j] - D[i,j] \geq 2$. Since by hypothesis, $D[i-1,j-1]$ can differ from $D[i-1,j]$ by at most 1, we must also have $D[i-1,j-1] - D[i,j] \geq 1$. Similarly, since $D[i,j-1]$ can differ from $D[i-1,j-1]$ at most 1, we must have $D[i,j-1] - D[i,j] \geq 0$. Thus, $D[i,j]$ is strictly less than the three values (i.e., $D[i,j]+c_{ij}$, $D[i-1,j]+1$, and $D[i,j-1]+1$) of which it is taken as a minimum. Contradiction.

Similarly, we can prove that our hypothesis holds for horizontal neighbors.

This completes the proof for lemma 3.2.

Now we can prove lemma 3.1.

*Proof of Lemma 3.1*:

Because of its definition, $D[i,j]$ cannot be greater than $D[i-1,j-1]+1$. We need only to show the second part: $D[i,j] \geq D[i-1,j-1]$.

The proof is by contradiction.

Suppose $D[i,j] < D[i-1,j-1]$. Then, $D[i,j]$ must have been obtained by adding 1 to either $D[i-1,j]$ or $D[i,j-1]$. Say it's $D[i-1,j]$. Then we must have

$$D[i-1,j] = D[i,j] - 1 \leq D[i-1,j-1] - 2 \ (*)$$

This is impossible by lemma 3.2. Contradiction.

*Lemma 3.3*: If the cost of a substitution is 2, then for every $i$ and $j$, $D[i,j]$ is in the range of $D[i-1,j-1]$ through $D[i-1,j-1]+2$.

*Proof of Lemma 3.3*:

Since $D[i,j]$ is the minimum of $D[i-1,j-1]+c_{ij}$, $D[i-1,j]+1$ and $D[i,j-1]+1$, and $D[i-1,j], D[i,j-1]$ cannot be greater than $D[i-1,j-1]$ by more than 1 (lemma 3.2), $D[i,j]$ cannot exceed $D[i-1,j-1]+2$.

On the other hand, if $D[i,j] < D[i-1,j-1]$, then, we must have $D[i,j] < D[i-1,j]+1$ and $D[i,j] < D[i,j-1]+1$ (lemma 3.2). That's in contradiction with the definition of $D[i,j]$ (as a minimum of three values).$\in$

Now that we have Lemma 1–3, we can derive the method of constructing matrix $D$ by using remainders modulo 4 arithmetic/comparisons for all integer variables in algorithm 4. That is, we compute only the least significant 2 bits of binary representations of $D[i,j]$'s. Thus, only 2 bits are needed for each integer variable.

*Theorem 1*: We can compute the remainder modulo four of each $D[i,j]$ by using the following rules. (All subtractions and additions are done modulo 4).

```
rule 1 (for $l$ =1):
  if $c = 0$ or $D[i-1,j] = D[i-1,j-1] -_{mod4} 1$ or
           $D[i,j-1] = D[i-1,j-1] -_{mod4} 1$
  then   $D[i,j] = D[i-1,j-1]$;
  else
     $D[i,j] = D[i-1,j-1] +_{mod4} 1$;
```

```
rule 2 (for $l$ =2):
  if $c = 0$ or $D[i-1,j] = D[i-1,j-1] -_{mod4} 1$ or
           $D[i,j-1] = D[i-1,j-1] -_{mod4} 1$
  then   $D[i,j]$ =D[i-1,j-1];
  else if $D[i-1,j] = D[i-1,j-1]$
  then   $D[i,j] = D[i-1,j-1] +_{mod4} 1$;
  else
     $D[i,j] = D[i,j-1] +_{mod4} 1$;
```

*Proof*:
For rule 1: By lemma 3.1, $D[i,j]$ must take the value of either $D[i-1,j-1]$ or $D[i-1,j-1]+1$. $D[i,j]$ will take the former value when either $c$ is 0 or $D[i-1,j] = D[i-1,j-1] -_{mod4} 1$, or $D[i,j-1] = D[i-1,j-1] -_{mod4} 1$, otherwise, $D[i,j]$ will take the latter value.

For rule 2: By lemma 3.3, $D[i,j]$ must take a value in the range of $D[i-1,j-1]$ through $D[i-1,j-1]+2$. $D[i,j]$ will take the first value when either $c_{ij}$ is 0 or $D[i-1,j] = D[i-1,j-1]-1$, or $D[i,j-1] = D[i-1,j-1] -_{mod4} 1$; Otherwise, it'll achieve the second value if $D[i-1,j] = D[i-1,j-1]$; If none of the above conditions are met, it'll achieve the second or third value depending on the value of $D[i,j-1]$.

Example:(assume $l = 1$)

Suppose $D[i-1,j-1]$, $D[i-1,j]$ and $D[i,j-1]$ are 3, 0, 3, respectively; Suppose $c_{ij} = 1$. Then, by rule 1, since $c_{ij} \neq 0$ and $D[i-1,j]$ or $D[i,j-1]$ does not equal to $D[i-1,j-1] -_{mod4} 1$, $D[i,j]$ should be $D[i-1,j-1] +_{mod4} 1$, that is, 0.

Of course, if we stop short at only knowing the remainder four of the edit distance, we are going nowhere. The trick is : we can reconstruct the true result by using an up-down counter at the end of our $PE$ array.

The output at $PE_m$ is the remainder modulo 4 of $D[m, t-m+1]$. In the $PE$ array architecture, we connect a counter to $PE_m$. Before $t = m+1$ (i.e., before the $PE$ array outputs $D[m, 1]$), we store a value $m$ (that is, the value of $D[m, 0]$) in the counter. At $t = m+1$, we get from $PE_m$ the output of remainder modulo 4 of $D[m, 1]$. By lemma 3.3, we know whether to increase or decrease the counter by 1 or let it remain the same by judging whether the output is greater or smaller than (in mod-4 sense) the remainder modulo 4 of the value in the counter. For example, if the current value in the counter is 5 (its remainder modulo 4 is 1), and the current output from $PE_m$ is 0, then, the value of the counter should be decreased to 4, because $0 <_{mod4} 1$ (one can also think of it this way: the new value should be 4, 5, or 6 by lemma 3.2, and only 4 has a zero remainder modulo 4). Thus, at time step $t > m$, the counter can always be properly adjusted in accordance with the mod-4 output from $PE_m$, giving the real output.

## 3.5 Discussion

(1) Although in the presentation of this algorithm, the physical dimension of the processor array is determined by the pattern size $m$, in a real hardware implementation, the physical dimension of $PE$ array could be made large enough to accommodate most real-life $m's$. Of course, the algorithm needs to be modified. For example, each $PE_i$ with $i > m$ may simply transfer whatever is received to its southern neighbor, thus propagating the results down to the lowest $PE$.
(2) This algorithm enables a VLSI implementation. Clearly, the $PE's$ can be very simple, with minimal memory, and minimal processing power.
(3) One possible commercial usage of this algorithm would be to fabricate cheap coprocessors implementing the algorithm. Such a coprocessor might be attached directly to a disk drive controller. This would enable application programs to do very fast approximate text search (ideally, at the same speed as that at which data is read from the disk surface). This kind of computing power is essential in many specialized applications.

As an indirect evidence of the usefulness of such a device, [Hollaar91] reviewed the development of disk-attached hardware-based pattern matchers for *exact string matching* problems. If, instead of attaching exact-pattern-matchers, we attach approximate-pattern-matchers (note again, when $k = 0$, our approximate-pattern-matcher reduces to exact-pattern-matcher) to hard disks, the functionality of such a device would be much more versatile, and meet with the needs of a wider range of applications.
(4) Our algorithm does not compute the optimal alignment path, i.e., the sequence of edit steps needed to achieve the (local) minimum distance between the pattern and some portion of the text. The user can always use other means to do this once the interesting segments of the text are located.

### 3.6 Architectural Alternatives – message-passing $PE$ network

Although the algorithm is described as systolic with global synchronization, this algorithm is also suitable for standard distributed message-passing architectures. Each $PE$ repetitively reads data from NORTH, does some computation, then sends data to its SOUTH channel. The synchronization can be guaranteed by blocked I/O, and it is trivial to prove that starting from a state in which every $PE$ is waiting to fetch data from NORTH, the time interval between two successive results from $PE_m$ cannot be greater than a small constant, which is determined by the time needed for computation and communication.

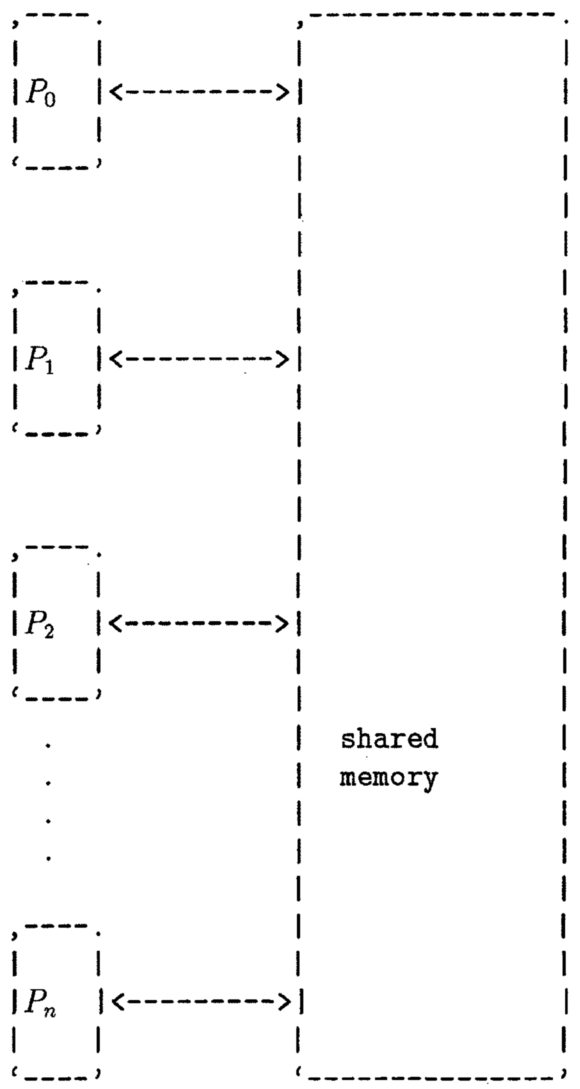
## 4 An $O(k)$ CRCW algorithm and related ideas

### 4.1 Introduction

In the study and research of parallel computing, abstract, theoretical computing models are often used in the literature. One important model is the parallel random-access machine, or PRAM in short.

In this section, we will present several ways of solving the *k-differences string-matching* problem in time $O(k)$, using PRAM model (more specifically, CRCW PRAM model, which will be explained below).

The following figure shows the general architecture of the PRAM model.



$P_i$ represents the $i_{th}$ processor.

That is, in a PRAM model, all processors can read from or write to a shared memory in parallel[Cormen90] [Leighton92] .

The key assumption concerning the PRAM model performance is that running time can be measured as the number of parallel memory accesses effected by an algorithm.

Different assumptions can be made in the PRAM model in regards to concurrent reads from or writes to the same memory location, which leads to different submodels within the PRAM paradigm.

Commonly used submodels are:

(1) EREW: exclusive read and exclusive write,
(2) CREW: concurrent read and exclusive write,
(3) ERCW: exclusive read and concurrent write, and
(4) CRCW: concurrent read and concurrent write.

See [Cormen90] for a detailed discussion of these variations of the PRAM model.

In this section we will be using CRCW PRAM model. That is, concurrent reads from and concurrent writes to the same memory location are allowed.

As stated in [Cormen90], "when multiple processors write to the same location in a CRCW algorithm, the effect of the parallel write is not well defined without additional elaboration."

Several assumptions have been used in the literature:

*a. Arbitrary.* The value written is arbitrarily chosen among the writers.

*b. Priority.* Store the value written by the lowest-indexed processor.

*c. Combination.* Store some specified combination of the values.

*d. Common−value.* When multiple processors write to the same location, they must write the same thing.

Different assumptions affect the complexity of our algorithm in different ways. This can be seen shortly.

## 4.2 $O(k)$ time with $O(m \times n)$ processors

First let's look back at algorithm 2 (Ukkonen's algorithm). For convenience, we restate algorithm 2 here.

*Algorithm* 2:

```
Initialization:
      for $0 \le d \le n$ do $L_{d,e} \leftarrow -1$
      for $-(k+1) \le d \le -1$ do $L_{d,|d|-1} \leftarrow |\, d \,| -1$, $L_{d,|d|-2} \leftarrow |\, d \,| -2$
      for $-1 \le e \le k$ do $L_{n+1,e} \leftarrow -1$
Main loop:
      for $e = 0$ to $k$ do
         for $d = -e$ to $n$ do
1           $row \leftarrow \max[\ (L_{d,e-1}+1), (L_{d-1,e-1}), (L_{d+1,e-1}+1)\ ]$
            $row \leftarrow \min(row, m)$
2           while $row < m$ and $row + d < n$ and $A_{row+1} = T_{row+1+d}$ do
                $row \leftarrow row + 1$
3           $L_{d,e} \leftarrow row$
4           if $L_{d,e} = m$ then
                print "There is an occurrence ending at $T_{d+m}$"
```

Since the inner for-loop can be done in parallel, if we can execute statement 2 (while-loop) in $O(1)$, then, the main loop can be done in $O(k)$. That's exactly the goal Landau and Vishkin [Landau89] have in mind when they developed their new algorithm (algorithm 3 in this thesis). But their use of a suffix tree causes significant overhead and resulted in a higher time complexity of $O(k + \log n)$.

Can the overall time complexity of $O(k)$ be achieved at all? This is an interesting theoretical question. And this is the goal of our efforts here.

Suppose we have $m \times n$ (again, $m$, $n$ are the lengths of pattern string $A$ and text string $T$, respectively) processors $P_{i,j}$, $(1 \le i \le m$, $1 \le j \le n)$, where $P_{i,j}$ looks at pattern element $A_i$ and text element $T_j$.

The objective is to speed up the inner while loop (statement 2 in algorithm 2).

Let *row_init* denote the initial value of the variable *row* computed by the max function just before the while loop. What the while loop does is essentially the following computation:

$$L_{d,e} \leftarrow \mathtt{min}\{row : row = m \;\mathtt{OR}\; (row \geq row\_init \;\mathtt{AND}\; A_{row+1} \neq T_{row+d+1})\} \qquad (1)$$

That is, the while loop goes down $D - diagonal\ d$, starting with the cell whose row number is *row_init*, until hitting a cell with a mismatch between its corresponding $A$ and $T$ elements. Then $L_{d,e}$ is set to the row number just before the mismatch.

In the above formula, this is expressed as a computation of finding a minimum among at most $m$ row numbers. More specifically, in our situation, since each cell on a $D - diagonal$ has a corresponding processor, in doing the minimum finding, we have, as resources, the same number of processors as the number of row numbers. Can this task of finding minimum among at most $m$ numbers be accomplished in $O(1)$ time?

We assume a CRCW PRAM model with assumption $b$ (the *priority* model). More specifically, take the row number $i$ of processor $P_{i,j}$ as its physical index. To determine the minimum value in formula (1), just let each processor concerned (i.e., each processor which is on $D - diagonal\ d$, has a physical index greater than *row_init*, and its next neighbor down the diagonal has a mismatch) write their index to $L_{d,e}$. In the *priority* model assumed, the value written has to be the minimum of those row indices.

Thus, computation in (1) can be done in $O(1)$ time trivially. One gets a simple CRCW $O(k)$ algorithm using $(n - m + k + 2) \times m$ processors.

The next algorithm implements the above idea.

*Algorithm* 7: (based on algorithm 2)

```
Initialization:

for $D - diagonal$ $d$, $0 \le d \le n$ parallel-do
        $L_{d,e} \leftarrow -1$
for $D - diagonal$ $d$, $-(k+1) \le d \le -1$ parallel-do
        $L_{d,|d| -1} \leftarrow |d| - 1$
        $L_{d,|d| -2} \leftarrow |d| - 2$
for $-1 \le e \le k$ parallel-do
        $L_{n+1,e} \leftarrow -1$

Main loop:

for $e = 0$ to $k$ do
  for $D - diagonal$ $d$, $-e \le d \le n$ parallel-do
      $row\_init_d \leftarrow$ max[$(L_{d,e-1} + 1), (L_{d-1,e-1}), (L_{d+1,e-1} + 1)$]
      $L_{d,e} \leftarrow$ min($row\_init_d$, $m$)
      if $L_{d,e} < m$ then
           for $P_{i,i+d}$, $row\_init_d \le i \le m$ parallel-do
              if ($i \neq m$) then
                if ($A_{i+1} \neq T_{i+1+d}$)
                      $L_{d,e} \leftarrow i$    //concurrent write
              else
                $L_{d,e} \leftarrow i$          //concurrent write
      if $L_{d,e} = m$ then
           print ''There is an occurrence ending at $T_{d+m}$''
```

As is said earlier, different assumptions of the CRCW model affect the complexity of our algorithm in different ways.

For example, if we assume assumption $a$, (i.e., *Arbitrary* model), it seems that there is no simple way to do the minimum-finding. Some kind of lock mechanism must be used to coordinate the writers, and the lock mechanism is inherently serial, the resulting time complexity cannot be $O(1)$ (It has to be some function of $m$).

The impact of assumption $d$, called common-CRCW model, to our algorithm will be discussed in next subsection.

Lastly, one can easily show that with assumption $c$, i.e., the *combination* model, one can also achieve the $O(1)$ minimum-finding goal in a similar way. To do so, one needs, within the *combination* model, to further assume that the combination value written will always be the minimum number among all the numbers being written (In [Cormen90], it is said that one typical way of combining the values is taking maximum).

## 4.3 $O(k)$ time with $O(n \times m \times m)$ processors

The same worst-case time complexity bound, $O(k)$, can also be achieved with assumption $d$, the common-CRCW model. But one has to use much more processors in this case.

For taking minimum among at most $m$ numbers, we can use an existing $O(1)$-time-complexity algorithm which uses $m \times m$ processors. See page 704 of [Cormen90]. They described a parallel algorithm which computes maximum among $m$ numbers in $O(1)$ time using $m \times m$ processors in a common-CRCW model. For our situation, we need $m \times m$ processors for each $D - diagonal$. This increases the number of processors needed by a factor of $m$. Thus, one gets an $O(k)$ CRCW algorithm that uses $(n - m + k + 2) \times m \times m$ processors.

Since this idea seems only have theoretical value, and since the resulting algorithm would be rather tedious, we will not try to give one here. But the idea should be quite clear.

## 4.4 Conclusion

PRAM model is a widely studied abstract parallel computing model. Although it omits many details which exist in actual architectures, it nevertheless grasps many key features of various parallel machines.

This section shows several ways in which one can get an $O(k)$ time algorithm for the *k difference approximate string matching problem*, each being associated with a specific CRCW model assumption.

Thus,the *k difference approximate string matching problem* CAN be solved in $O(k)$ time in CRCW PRAM computation model.

## 5 Acknowledgement

This research work is supervised, guided by my graduate advisory committee, which consists of Prof. Alden H. Wright, Prof. Ronald E. Wilson in Computer Science department, and Prof. Ralph Judd in Division of Biological Sciences.

Thanks especially go to committee chairman, Prof. Alden H. Wright. This research is actually the result of our joint effort.

## References

[Apostolico88] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, Parallel construction of a suffix tree with applications, *Algorithmica*, **3**, (1988), 347–365.

[Cormen90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, "Introduction to Algorithms" (chapt 30, algorithms for parallel computers) MIT press, 1990.

[Hollaar91] L. A. Hollaar, Special-purpose hardware for text searching: past experience, future potential, *Information Processing & Management*, **27**, No. 4 (1991), 371–378.

[Galil86] Z. Galil, and R. Giancarlo, Improved string matching with k mismatches, *SIGACT News*, **17** (1986), 52–54.

[Galil88] Z. Galil, and R. Giancarlo, Data structures and algorithms for approximate string matching, *J. Complexity*, **4** (1988), 33–72.

[Galil90] Z. Galil, and K. Park, An improved algorithm for approximate string matching, *SIAM J. Comput.*, **19**, No. 6 (1990), 989–999.

[Guibas79] L. J. Guibas, H. T. Kung, and C. D. Thompson, Direct VLSI implementation of combinatorial algorithms, *Proc. Conf. VLSI: Architecture, Design, and Fabrication,* California Institute of Technology, Pasadena, CA, 1979, pp. 509-525.

[Landau85] G. M. Landau, and U. Vishkin, Efficient string matching in the presence of errors, *in* "Proceedings, 26th IEEE FOCS, 1985," pp. 126–136.

[Landau87] G. M. Landau, B. Schieber, and U. Vishkin, Parallel construction of a suffix tree, *in* "Proceedings 14th ICALP,", Lecture Notes in Computer Science, **267**,(1987), 314–325.

[Landau88] G. M. Landau, and U. Vishkin, Fast string matching with k differences, *J. Comput. System Sci.*, **37** (1988), 63–78.

[Landau89] G. M. Landau, and U. Vishkin, Fast parallel and serial approximate string matching, *J. Algorithms*, **10** (1989), 157–169.

[Lander91] E. S. Lander, R. Langridge, and D. M. Saccocio, Mapping and interpreting biological information, *Communications of The ACM*, **34**, No. 11 (1991), 33–39.

[Leighton92] F. Thomson Leighton, "Introduction to Parallel Algorithms and Architectures" (Section 3.6, Simulating a Parallel Random Access Machine) Morgan Kaufmann Publishers, 1992.

[Lipton85] R. J. Lipton, and D. Lopresti, A systolic array for rapid string comparison, *in* "Proceedings of 1985 Chapel Hill Conference on VLSI," pp. 363–376.

[Manber89] U. Manber, "Introduction to Algorithms, a creative approach," Addison-Wesley, 1989, pp. 155–158.

[Needleman70] S. B. Needleman, and C. D. Wunsch, A general method applicable to the search for similarities in the amino acid sequences of two proteins, *J. Mol. Bio.*, **48** (1970), 444–453.

[Sankoff83] D. Sankoff, J. B. Kruskal (Eds.), "Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison", Addison-Wesley, Reading, MA, 1983.

[Sellers74] P. H. Sellers, An algorithm for the distance between two finite sequences, *J. Combinator. Theor.*, A16 (1974), 253–258.

[Sellers79] P. H. Sellers, Pattern recognition in genetic sequences, *Pro. Natl Acad. Sci. USA*, **76** (1979), 3041.

[Ukkonen83] E. Ukkonen, On approximate string matching, *in* "Proceedings Int. Conf. Found. Comput. Theory," *Lecture Notes in Computer Science*, **158** (1983), 487–495.

[Ukkonen85a] E. Ukkonen, Finding approximate pattern in strings, *J. Algorithms*, **6** (1985), 132–137.

[Ukkonen85b] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control*, **64** (1985), 100–118.

[Wagner74] R. A. Wagner, and M. J. Fischer, The string-to-string correction problem, *J. ACM*, **21** (1974), 168–173.