Efficient Parallel And Serial Approximate String Matching

by

Gad M. Landau[1, 3]

Uzi Vishkin[2, 3]

Ultracomputer Note #101

Computer Science Department Technical Report #221

February, 1986

[1] Department of Computer Science, School of Mathematical Sciences, Tel Aviv University

[2] Department of Computer Science, Courant Institute of Mathematical Sciences, New York University and (present address) Department of Computer Science, School of Mathematical Sciences, Tel Aviv University. The research of this author was supported by NSF grants NSF-DCR-8318874 and NSF-DCR-8413359 and ONR grant N00014-85-K-0046.

[3] The research of both authors was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy under contract number DE-AC02-76ER03077.

ABSTRACT

Consider the string matching problem, where differences between characters of the pattern and characters of the text are allowed. Each difference is due to either a mismatch between a character of the text and a character of the pattern or a superfluous character in the text or a superfluous character in the pattern. Given a text of length *n*, a pattern of length *m* and an integer *k*, we present parallel and serial algorithms for finding all occurrences of the pattern in the text with at most *k* differences. The first part of the parallel algorithm consists of analysis of the pattern and takes $O(log\ m)$ time using m^2 processors. The rest of the algorithm consists of handling the text. The text handling part applies the following new approach. This part starts by obtaining a *concise* characterization of the text which is based solely on substrings of the pattern in $O(log\ m)$ time using $n/log\ m$ processors. Then the desired output is derived from this characterization together with the tables built in the first part in $O(k)$ time using *n* processors.

The serial algorithm follows also this new approach for handling the text. It runs in $O(kn)$ time for alphabet whose size is fixed. For general input the algorithm requires $O(n(k + \log m))$ time. In both cases the space requirement is $O(n)$.

1. Introduction

The problem. *Input*. Two arrays: $A = a_1,...,a_m$ - the pattern, $T = t_1,...,t_n$ - the text and an integer *k* (≥ 1).

In the known problem of pattern matching in strings (e.g., as discussed in [KMP-77]) we are interested in finding all occurrences of the pattern in the text. In the present paper we are interested in designing an algorithm that finds all such occurrences with at most *k* differences.

Example. Let the text be *abcdefghi* , the pattern *bxdyegh* and $k=3$. Let us see whether there is an occurrence with ≤ *k* differences that ends at the eighth location of the text. For this we propose the following correspondence between *bcdefghi* and *bxdyegh.* 1. *b* (of the text) corresponds to *b* (of the pattern).2. *c* to *x*. 3. *d* to *d*. 4. Nothing to *y*. 5. *e* to *e*. 6. *f* to nothing. 7. *g* to *g*. 8. *h* to *h*. The correspondence can be illustrated as

```
b x d y e   g h
b c d   e f g h i
```

In only three places the correspondence is between non-equal characters, implying that there is an occurrence of the pattern that ends at the eighth location of the text with 3 differences as required.

We distinguish three types of differences:

(a) A character of the pattern corresponds to a different character of the text. (Item 2 in the Example). In this case we say that there is a *mismatch* between the two characters.

(b) A character of the pattern corresponds to "no character" in the text. (Item 4).

(c) A character of the text corresponds to "no character" in the pattern. (Item 6).

We consider the following problem.

The *string matching with k-differences* problem. (In short, the $k-differences$ problem).

Find all occurrences of the pattern in the text with at most k differences of type (a),(b) and (c).

The case $k = 0$ in the k-differences problem is the extensively studied string matching problem. There are a few notable algorithms for the string matching problem: linear time serial algorithms - [BM-77], [GS-83], [KMP-77], [KR-80] (a randomized algorithm) and [V-85b], parallel algorithms [G-84] and [V-85b].

Even these parallel algorithms for exact string matching had to abandon their preceding linear time serial algorithms since these serial algorithms do not seem amenable to parallelism. We note that none of these serial and parallel algorithms is suitable to cope with the k-differences problem. Moreover, the remark below explains why even the way by which parallelism is approached in these parallel algorithms is unlikely to be generalizable for approximate string matching.

Remark. [G-84] and [V-85b] gave parallel algorithms for exact string matching. We give a short description of their approach and explain why we had difficulties in applying it for the k-differences problem. The main part in the text handling parts of each of these algorithms consists of eliminating many entries of the text for which occurrences of the pattern cannot start. This elimination process iterates the following step: it picks a proper pair of "close" entries which have not yet been eliminated. The pair is proper in the sense that, based on information gathered in the pattern analysis, an occurrence may start in at most one of these entries. Then, one of these entries is eliminated. This results in a small enough number of remaining entries in which occurrences may start. (The final part in each of these algorithms is a straightforward procedure which checks whether there is an occurrence in each of these remaining entries.) We see no way for applying a similar elimination process for approximate string matching problems. The reason being that the differences which are allowed between the pattern and the text enable coexistence of seemingly contradicting occurrences. Indeed, our solution is constructive in the sense that it finds all occurrences without a preceding stage in which some entries in which an occurrence is impossible are eliminated.

The model of computation used in this paper is the random-access-machine (RAM) [AHU-74] for the serial algorithm, and the concurrent-read exclusive-write (CREW) parallel random access machine (PRAM) for the parallel algorithm. A PRAM employs p synchronous processors all having access to a common memory. A CREW PRAM allows simultaneous access by more than one processor to the same memory location for read but not for write purposes. See [V-83] for a survey of results concerning PRAMs.

The k-differences problem is not only a basic theoretical problem. It also has a strong pragmatical flavor. In practice, we often need to analyze situations where the data is not completely reliable. Specifically, consider a situation where the strings which are the input for our problem contain errors and we still need to find all possible occurrences of the pattern in the text as in reality. The errors may include a character being replaced by another character, a character being omitted or a superfluous character being inserted. Assuming some bound on the number of errors would clearly imply our problem. We refer the reader to [SK-83], a book which is essentially devoted to various instances of the k-differences problem.

The book gives a comprehensive review of applications of the problem in a variety of fields, including: computer science, molecular biology and speech recognition.

We give a first parallel algorithm for the *k*-differences problem. The algorithm has three main parts: I. Analysis of the pattern. II. Analysis of the text. III. Finding all occurrences of the pattern in the text with at most *k* differences.

Part I processes the pattern only and results in a few tables. These tables contain information on how substrings of the pattern relate to other substrings of the pattern. In principle, similar constructs were used in early string matching algorithms like [KMP-77]. Part I needs $O(log\ m)$ time using m^2 processors.

Part II processes the text using the tables that were built in Part I. It results in a table which characterizes the whole text using only substrings of the pattern as yardsticks. Such constructs seem to be new.

Part III uses only the tables built in Parts I and II in order to derive the desired output. In other words, the characterization of the text which was obtained in Part II turns out to be so powerful that we do not need to take another look at the text.

The complexity results for the text handling parts (Parts II and III) demonstrate that the characterization obtained in Part II is concise: (a) The characterization can be computed efficiently - Part II needs $O(log\ m)$ time using $n/log\ m$ processors to compute it. (b) The characterization provides for an efficient solution of the *k*-differences problem - Part III needs $O(k)$ time using *n* processors for finishing the solution of the *k*-differences problem.

This present paper demonstrates how parallel algorithms can enrich the field of serial algorithms. We first discovered the parallel algorithm. We then noticed that the parallel algorithm yields as a byproduct a new serial algorithm for the *k*-differences problem which is considerably simpler. The serial algorithm runs in $O(kn)$ time for alphabet whose size is fixed and requires $O(n(k + logm))$ time for general input. In both cases the space requirement is $O(n)$. In [LV-85a], [LV-85c] the authors give two implementations of a serial algorithm for the *k*-differences problem. The first one [LV-85a] runs in $O(m^2 + nk^2)$ for general input and requires $O(m^2)$ space. The second one [LV-85c] runs in $O(m + k^2 n)$ time for alphabet whose size is fixed. For general input the algorithm requires $O(m\log\ m + k^2 n)$ time. In both cases the space requirement is $O(m)$. Our new serial algorithm is faster than these previous algorithms when the size of the alphabet is fixed or for general input when k^2 is larger than log *m* by order of magnitude.

Using notations of the first paragraph of this section we define the *k-mismatches* problem as follows. The input is the same as for the *k*-differences problem. The problem is to find all occurrences of the pattern in the text with at most *k* differences of type (a). [LV-85a], [LV-85b] give an algorithm for the *k*-mismatches problem. It runs in $O(k(mlogm + n))$ time for general input. Our new serial algorithm for the *k*-differences can handle also the *k*-mismatches problem. So, when the alphabet size is fixed or for general input when *kmlogm* is larger than *nlogm* by order of magnitude then the new algorithm is better than [LV-85a], [LV-85b].

In the recent survey on future directions for research in string matching [G-85], the *k*-mismatches problem is discussed. An open question which is proposed in the paper is whether the simple (dynamic programming) algorithm for the k-mismatches problem which takes $O(mn)$ serial time can be improved. A similar $O(mn)$ time algorithm solves the k-differences problem. We note that the present paper answers affirmatively this question also for the k-differences problem which seems more general.

The serial algorithm is given in Section 2. In order to make the presentation more intuitive Part III of the parallel algorithm is described in Section 3. Part II in Section 4 and Part I in Section 5.

2. The Serial Algorithm

In this section we give our new serial algorithm for the k-differences problem. As a warm up we start with two known serial $O(mn)$ time algorithms for this problem. The first one is a simple dynamic programming algorithm. The second algorithm follows the same dynamic programming computation in slightly different way which will help to understand the new algorithm. Subsection 2.3 gives the new serial algorithm.

2.1. The dynamic programming algorithm.

We use a matrix $D_{[0,...,m;0,...,n]}$, where $D_{i,l}$ is the minimum number of differences between $a_1, \ldots, a_i$ and any successive substring of the text ending at t_l.
It should be obvious that if $D_{m,l} \leq k$ then there must be an occurrence of the pattern in the text with at most k differences that ends at t_l.

The following algorithm computes the matrix $D_{[0,...,m;0,...,n]}$

Initialization *for all* l, $0 \leq l \leq n$, $D_{0,l} := 0$
for all i, $1 \leq i \leq m$, $D_{i,0} := i$

for i:=1 to m do
for l:=1 to n do

$D_{i,l}$:= min ($D_{i-1,l}+1$, $D_{i,l-1}+1$, $D_{i-1,l-1}$ if $a_i = t_l$ or $D_{i-1,l-1}+1$ otherwise)
($D_{i,l}$ is the minimum of three numbers. These three numbers are obtained from the predecessors of $D_{i,l}$ on its column, row and diagonal, respectively)

Complexity. The algorithm clearly runs in $O(mn)$ time.

2.2. An alternative dynamic programming computation.

The description reminds to some extent [U-83]. It computes the matrix D, of the dynamic programming algorithm, using its diagonals. A *diagonal* d of the matrix consists of all $D_{i,l}$'s such that $l-i = d$.

For a number of differences *e* and a diagonal *d*, let $L_{d,e}$ denote the largest row *i* such that $D_{i,l} = e$ and $D_{i,l}$ is on diagonal *d*. The definition of $L_{d,e}$ clearly implies that *e* is the minimum number of differences between $a_1, \ldots, a_{L_{d,e}}$ and any substring of the text ending at $t_{L_{d,e}+d}$. It also implies that $a_{L_{d,e}+1} \neq t_{L_{d,e}+d+1}$. For our *k*-differences problem we need only the values of $L_{d,e}$*'s*, where *e* satisfies $e \leq k$.

If one of the $L_{d,e}$'s equals *m*, for $e \leq k$, it means that there is an occurrence of the pattern in the text with at most *k* differences that ends at t_{d+m}.

We compute the $L_{d,e}$'s by induction on *e*. Given *d* and *e* we show how to compute $L_{d,e}$ using its definition. Suppose that for all $x < e$ and all diagonals y $L_{y,x}$ was already computed. Suppose $L_{d,e}$ should get the value *i*. That is, *i* is the largest row such that $D_{i,l} = e$, and $D_{i,l}$ is on the diagonal *d*. The algorithm of the previous subsection reveals that $D_{i,l}$ could have been assigned its value *e* using one (or more) of the following data:

(a) $D_{i-1,l-1}$ (which is the predecessor of $D_{i,l}$ on the diagonal *d*) is $e-1$ and $a_i \neq b_l$. Or, $D_{i,l-1}$ (the predecessor of $D_{i,l}$ on row *i* which is also on the diagonal "below" *d*) is $e-1$. Or, $D_{i-1,l}$ (the predecessor of $D_{i,l}$ on column *l* which is also on the diagonal "above" *d*) is $e-1$.

(b) $D_{i-1,l-1}$ is also *e* and $a_i = b_l$.

This implies that we can start from $D_{i,l}$ and follow its predecessors on diagonal *d* by possibility (b) till the first time possibility (a) occurs.

The following algorithm "inverses" this description in order to compute the $L_{d,e}$'s. $L_{d,e-1}$, $L_{d-1,e-1}$, and $L_{d+1,e-1}$ are used to initialize the variable *row*, which is then increased by one at a time till it hits the correct value of $L_{d,e}$.

The following algorithm computes the $L_{d,e}$*'s*

Initialization *for all* d, $0 \leq d \leq n+1$, $L_{d,-1} := -1$
for all d, $-(k+1) \leq d \leq -1$ *do*
$L_{d,|d-2|} := -\infty$
$L_{d,|d-1|} := |d-1|$

2. *for e:=0 to k do*
for d:=-e to n do
3. $row := max[(L_{d,e-1}+1), (L_{d-1,e-1}), (L_{d+1,e-1}+1)]$
4. *while* $a_{row+1} = t_{row+1+d}$ *do*
$row := row + 1$
5. $L_{d,e} := row$
6. if $L_{d,e} = m$ *then*
print *THERE IS AN OCCURRENCE ENDING AT t_{d+m}*

Remarks. a) For every i,l, $D_{i,l} - D_{i-1,l-1}$ is either zero or one. b) The values of the matrix *D* on diagonals *d*, such that $d > n-m+k+1$ and $d < -k$ are useless for the solution of the *k*-differences problem.

Correctness of the algorithm.

Claim. $L_{d,e}$ gets its correct value.

Proof of claim. By induction on *e*.

Let $e = 0$. Consider the computation of $L_{d,0}$, $(d \geq 0)$. Instruction 3 starts by initializing *row* to 0. Instructions 4 and 5 find that $a_1, \ldots, a_{L_{d,0}}$ is equal to $t_{d+1}, \ldots, t_{d+L_{d,0}}$ and $a_{L_{d,0}+1} \neq t_{d+L_{d,0}+1}$. Therefore $L_{d,0}$ gets its correct value.

Let $e = l$. Assume that all $L_{d,l-1}$ are correct. (The reader can easily check that $L_{-l,l-1}$ and $L_{-l-1,l-1}$ get correct values in the Initialization - this should have actually been part of establishing the base of the induction.) Consider the computation of $L_{d,e}$, $(d \geq -e)$. Following Instruction 3 *row*, is the largest row on diagonal *d* such that $D_{row,d+row}$ can get value *e* by possibility (a). Then Instruction 4 finds $L_{d,e}$.

Complexity. We evaluate $L_{d,e}$'s for $n+k+1$ diagonals. For each diagonal the variable *row* can get at most *m* different values. Therefore, the computation takes $O(mn)$ time.

2.3. The new algorithm

The new algorithm has two steps:

Step I. Concatenate the text and the pattern to one string $(t_1, \ldots, t_n ¢ a_1, \ldots, a_m)$. Compute the suffixes tree of this string.

A methodological remark. Step I of the serial algorithm presented here combined Parts I and II of the parallel algorithm that follows. Step II corresponds to Part III.

Step II. Find all occurrences of the pattern in the text with at most *k* differences.

2.3.1. Step I.

Let us define the *suffixes tree* of a string $C = c_1, \ldots, c_l$:

1) It is a tree in which all the edges of the tree are directed away from the root. The out degree of each node of the tree is either zero (if the node is a leaf) or ≥ 2.

2) Each suffix $C_i = c_{i+1}, \ldots, c_l$ of the string defines a *leaf* of the tree. (The tree has *l* leaves.)

3) Let C_i and C_j be any two suffixes. Suppose $c_{i+1},...,c_{i+f}$ is their longest equal prefix. That is, $c_{i+1}, \ldots, c_{i+f}$ equals to $c_{j+1}, \ldots, c_{j+f}$ and $c_{i+f+1} \neq c_{j+f+1}$. Then, $c_{i+1}, \ldots, c_{i+f}$ defines an *internal node* (i.e., a node which is not a leaf) of the tree. Let *D* be a successive substring of the string *C*. Let *B* be a proper prefix of *D*. Suppose also that both *D* and *B* define nodes of the tree. Then there is an *edge* connecting the nodes of *D* and *B* if there is no successive substring *F* of the pattern such that the following three condition hold at once: *F* is a proper prefix of *D*, *B* is a proper prefix of *F* and *F* defines a node of the tree.

4) The substrings of two sibling edges (edges emanating from the same vertex of the tree) cannot have identical (nonempty) prefixes.

Upon construction of the suffixes tree we require that for each node *v* of the tree a successive substring $c_{i+1}, \ldots, c_{i+f}$ which defines it will be stored as follows: $START(v) := i$ and $END(v) := f$.

Remark. Up to isomorphism (of graphs) there is only one suffixes tree for a given string.

EXAMPLE. Given the string *abab$* the suffixes tree is:

$START(A) = 2$, $END(A) = 2$, $START(B) = 3$, $END(B) = 1$, $START(C) = 5$, $END(C) = 0$, $START(D) = 0$, $END(D) = 4$, $START(E) = 2$, $END(E) = 2$, $START(F) = 1$, $END(F) = 3$, $START(G) = 3$, $END(G) = 1$.

The suffixes tree

We compute the suffixes tree of the string $t_1, \ldots, t_n ¢\, a_1, \ldots, a_m$ using the serial algorithm of Weiner [W-73]. Actually, we replace each character of the text which does not appear in the pattern by the same special character Y and compute the suffixes tree for this string.

Complexity. [W-73] computes the suffixes tree in $O(n)$ time when the size of the alphabet is fixed. This is also the running time of Step I for fixed size alphabet. If the alphabet of the pattern contains x letters then it is easy to adapt this algorithm of [W-73] and Step I to run in time $O(n \log x)$. In both cases the space requirement of Step I is $O(n)$. (The reader is also referred to [CS-85] in which a lucid presentation of the algorithm of [W-73] is given).

2.3.2. Step II.

The matrix D and the $L_{d,e}$'s are exactly as in the alternative dynamic programming algorithm. We use this alternative algorithm with a very substantial change. Introducing this change in the present step of the algorithm and enabling it by proper preparation in the previous step is the main contribution of this paper in both the serial and parallel algorithms. The change is in Instruction 4, where instead of increasing variable *row* by one at a time until it reaches $L_{d,e}$, we find $L_{d,e}$ in $O(1)$ time!

For a diagonal d, the situation following Instruction 3 is that we matched (with e differences) $a_1, \ldots, a_{row}$ of the pattern with some substring of the text that ends at t_{row+d}. We want to find the largest q for which $a_{row+1}, \ldots, a_{row+q}$ equals $t_{row+d+1}, \ldots, t_{row+d+q}$. Let $LCA_{row,d}$ be the lowest common ancestor (in short LCA) of the leaves of the suffixes $t_{row+d+1}, \ldots$ and $a_{row+1}, \ldots$ in the suffixes tree. The desired q is simply $END(LCA_{row,d})$. Thus, the problem of finding this q is reduced to finding $LCA_{row,d}$. We use the algorithm of [HT-84] for the purpose of computing LCA's in the suffixes tree when ever we need to find such a q throughout the algorithm.

Complexity. Using the classification of [HT-84] we are interested in the *static lowest common ancestors* problem, where the tree is static but queries for lowest common ancestors of pair of vertices are given on line. That is, each query must be answered before the next one is known. The suffixes tree has $O(n)$ nodes. The algorithm of [HT-84] proceeds as follows. It preprocesses the suffixes tree in $O(n)$ time. Then, given $\alpha \geq n$ LCA queries it responds to them in a total of $O(\alpha)$ time. For each of the $n+k+1$ diagonals, we evaluate $k+1$ $L_{d,e}$'s. Therefore, we have $O(kn)$ LCA Queries. It will take $O(kn)$ time to process them. This time dominates the running time of Step II.

Complexity of the serial algorithm. The total time for the serial algorithm is, $O(kn)$ time for alphabet whose size is fixed and $O(n(k + logm))$ time for general input.

3. Parallel Algorithm - Finding All Occurrences Of The Pattern In The Text With At Most *k* Differences (Part III).

This section is devoted to the last part of the parallel algorithm. The presentation will clarify which information became available as a result of Parts I and II. The matrix D and the $L_{d,e}$'s are exactly as in the serial algorithm. Part III of the parallel algorithm employs $n+k+1$ processors. Each processor is assigned to a diagonal d, $-k \leq d \leq n$. The parallel treatment of the diagonals is the source of parallelism in Part III of our new algorithm.

For a diagonal d the situation following Instruction 3 is that we matched (with e differences) $a_1,...,a_{row}$ of the pattern with some substring of the text that ends at t_{row+d}. We want to find the largest q for which $a_{row+1},...,a_{row+q}$ equals $t_{row+d+1},...,t_{row+d+q}$.
In the serial algorithm we got this q from the suffixes tree. In the parallel algorithm we get q in a different way. We use two kinds of information from the previous parts of the algorithm:
a) An index g of the pattern which brings l to a maximum in the following match: $t_{row+d+1}, \ldots, t_{row+d+l} = a_{g+1}, \ldots, a_{g+l}$. (There is no larger l (and g) for which such a match holds.) This information was computed in Part II into an array called $BEST-FIT$ (see section 4).
b) The length f of the longest match between a_{row+1},. . . and a_{g+1},. . .. That is, $a_{row+1}, \ldots, a_{row+f} = a_{g+1}, \ldots, a_{g+f}$ and $a_{row+f+1} \neq a_{g+f+1}$. This information was computed in Part I into array $MAX-LENGTH$ (see section 5).

Observation. The desired q is the minimum between f and l. proof of the observation is straightforward.

We use the parameter d and the *pardo* command for the purpose of guiding each processor to its instruction.

Part III of the parallel algorithm

1. Initialization (as above)
2. *for e:=0 to k do*

for d:=-e to n pardo

3. $row := max[(L_{d,e-1}+1), (L_{d-1,e-1}), (L_{d+1,e-1}+1)]$
4. $L_{d,e} := row + min(f,l)$
5. if $L_{d,e} = m$ *then*

print *THERE IS AN OCCURRENCE ENDING AT t_{d+m}*

Complexity of Part III. We employs $n+k+1$ processors (one per diagonal). Each processor computes at most $k+1$ $L_{d,e}$'s. Obtaining the information from Parts I and II to compute each $L_{d,e}$ takes $O(1)$ time. Therefore, Part 3 takes $O(k)$ time using $n+k+1$ processors. Simulating the algorithm by n processors, instead of $n+k+1$ still gives $O(k)$ time.

4. Parallel Algorithm - Computation Of Array Best-Fit (Part II).

In this section we compute the one dimensional array $BEST-FIT[0,...,n-1]$. $BEST-FIT(i) = (g,l)$ means that $t_{i+1},...,t_{i+l} = a_{g+1}, \ldots, a_{g+l}$, and there is no larger l, such that there exists g, for which such a match holds. In this case we denote $BEST-FIT(i).1 = g$ and $BEST-FIT(i).2 = l$. The computation relies on the following information which was gathered in Part I of the parallel algorithm (see section 5):

1. The array $LOCATION[1,...,\beta]$, $LOCATION(i) = f$ means that character i in the alphabet appears in location f of the pattern ($a_f = i$).
2. The three dimensional array $PAIR-FIT[0,...,m-1;0,...,m-1;0,..., \lceil \log m \rceil]$. $PAIR-FIT(j,l,i) = (\lambda, f)$ means that $a_{j+1},...,a_{j+2^i}$ $a_{l+1},...,a_{l+f} = a_{\lambda+1}, \ldots, a_{\lambda+2^i+f}$ and there is no larger f, such that there exists λ, for which such a match holds.

Step 1. Using array *LOCATION* find for each character in the text a location in the pattern in which the same character appears (if there is one).

Note that Step 1 results in a characterization of the text using characters of the pattern only. Each stage of Steps 2 and 3 refines this characterization until the ultimate characterization is reached and entered into $BEST-FIT$.

Steps 2 and 3 use the scheme of parallel prefix sum computation in which a balanced binary tree guides the computation. We refer the reader to [V-84] for a detailed description. (For an earlier reference to using the scheme of parallel prefix sum computation see [FL-80].) The balanced binary tree is defined as follows: Each pair $[i,j]$, where $0 \le i \le \lceil \log n \rceil$, $0 \le j \le n-1$ and j is divisible by 2^i, defines a node of the binary tree whose *left son* is $[i-1,j]$ and *right son* is $[i-1,j+2^{i-1}]$.

Step 2. The computation proceeds in $\lceil \log m \rceil$ stages.

The output of stage i, $1 \le i \le \lceil \log m \rceil$: Essentially, it is the same as the input of stage $i+1$. For each j, $0 \le j \le n-1$ which is divisible by 2^i, we are given an index f of the pattern such that $t_{j+1}, \ldots, t_{j+2^i} = a_{f+1}, \ldots, a_{f+2^i}$, if such f exists. If such f does not exist we are given an index g of the pattern which brings l to a maximum in the following match: $t_{j+1}, \ldots, t_{j+l} = a_{g+1}, \ldots, a_{g+l}$.

The relation to the binary tree is clear: The "active" nodes at stage i are of the form $[i,j]$. The input that such a node needs can be obtained from its two sons. We further observe (and leave it to the reader to fill in the details) that using $PAIR-FIT$ a single processor can perform the operations corresponding to each node in $O(1)$ time. We refer to the computation at each node as *an operation.*

Observe that the computation starts at the leaves and moves towards the root of the balanced binary tree which guides the computation until it reaches the nodes of the tree whose distance from the leaves is $\lceil \log m \rceil$. The tree has n leaves and therefore $O(n)$ nodes. The $\lceil \log m \rceil$ stages use only the $\lceil \log m \rceil$ lower levels of the tree. Hence, this description implies a total of $O(n)$ operations and $O(log\ m)$ time.

Step 3 consists also of $\lceil \log m \rceil$ stages. Essentially, they amount to reversing the $\lceil \log m \rceil$ stages of Step 2. That is, the computation of Step 3 starts at nodes of the tree whose distance from the leaves is $\lceil \log m \rceil$ and ends at the leaves.

Step 3. The computation proceeds in $\lceil \log m \rceil$ stages. We describe stage i, $1 \le i \le \lceil \log m \rceil$. Let $\delta := \lceil \log m \rceil - i$.

The input which is relevant to stage i:

a) For each j, $0 \le j \le n-1$, which is divisible by $2^{\delta+1}$, we are given an index g of the pattern which brings l to a maximum in the following match: $t_{j+1}, \ldots, t_{j+l} = a_{g+1}, \ldots, a_{g+l}$. (That is, we have already found that $BEST-FIT(j) = (g,l)$.)

b) For each h, $0 \le h \le n-1$, which is divisible by 2^{δ} but is not divisible by $2^{\delta+1}$, we are given an index f of the pattern such that $t_{h+1}, \ldots, t_{h+2^{\delta}} = a_{f+1}, \ldots, a_{f+2^{\delta}}$, if such f exists. If such f does not exist we are given an index g of the pattern which brings l to a maximum in the following match: $t_{h+1}, \ldots, t_{h+l} = a_{g+1}, \ldots, a_{g+l}$; (observe, that in this case (g,l) is already the desired output for $BEST-FIT(h)$).

The result of stage i:

For each h, $0 \le h \le n-1$, which is divisible by 2^{δ} but is not divisible by $2^{\delta+1}$, we find $BEST-FIT(h)$. That is, we want to find an index g of the pattern which brings l to a maximum in the following match: $t_{h+1}, \ldots, t_{h+l} = a_{g+1}, \ldots, a_{g+l}$. This computation takes place at node $[\delta,h]$ of the tree. Finally, we observe that using $PAIR-FIT$ a single processor can carry out the computation required at each node, in $O(1)$ time.

Similar to Step 2, these log m stages require a total of $O(n)$ operations and $O(log\ m)$ time.

Complexity of Part II. We had a total of $O(n)$ operations and $O(log\ m)$ time. No difficulty will arise in applying the simulation scheme due to [Br-74]. Applications of this scheme for similar purpose were given in a few parallel algorithms. For instance, see [V-85b]. This will result in $O(log\ m)$ time using $n/\log m$ processors.

5. Parallel Algorithm - Analysis Of The Pattern (Part I).

5.1. Computation Of Array Max-Length

The input is the pattern $a_1, \ldots, a_m$. The output is the two dimensional array $MAX-LENGTH[0,...,m-1;0,...,m-1]$. $MAX-LENGTH(i,j)=f$ means that $a_{i+1}, \ldots, a_{i+f} = a_{j+1}, \ldots, a_{j+f}$, and $a_{i+f+1} \neq a_{j+f+1}$. In words, consider laying the suffix of the pattern starting at a_{i+1} over the suffix of the pattern starting at a_{j+1}. $MAX-LENGTH(i,j)$ is the longest match of prefixes between these two suffixes.

The pair (i,j), $0 \leq i,j \leq m-1$ is defined to be on *diagonal* d of $MAX-LENGTH$ if $j-i=d$, where possible values of d are $-(m-1) \leq d \leq m-1$. It is easy to see that: $MAX-LENGTH(i,j) = MAX-LENGTH(i+1,j+1) + 1$ if $a_{i+1} = a_{j+1}$ and $MAX-LENGTH(i,j) := 0$ otherwise. We compute $MAX-LENGTH$ in two steps:

a) *Initialization*: for each pair *(i,j)* $0 \leq i, j \leq m-1$ $MAX-LENGTH(i,j) := 1$ if $a_{i+1} = a_{j+1}$ and $MAX-LENGTH(i,j) := 0$ otherwise.

b) Using a parallel prefix sum computation we compute the values of each diagonal d of $MAX-LENGTH$ separately. (We compute the sum till the first 0 and not till the bottom of the diagonal.)

Complexity. In step a we have $O(m^2)$ operations and $O(1)$ time. In step b we have a total of $O(m)$ operations and $O(log\ m)$ time per diagonal. Applying the simulation scheme due to [Br-74] will result in $O(log\ m)$ time using $m^2/\log m$ processors.

5.2. Computation Of Array Pair-Fit

The input is the pattern $a_1, \ldots, a_m$. The output is the three dimensional array $PAIR-FIT[0,...,m-1;0,...,m-1;0,..., \lceil \log m \rceil]$.

$PAIR-FIT(j,l,i) = (\lambda, f)$ means that $a_{j+1},...,a_{j+2^i}\ a_{l+1},...,a_{l+f} = a_{\lambda+1}, \ldots, a_{\lambda+2^i+f}$ and there is no larger f (and λ) for which such a match holds. In words, concatenate to $a_{j+1},...,a_{j+2^i}$ the suffix of the pattern starting at a_{l+1}. As a result we get the concatenated string $a_{j+1},...,a_{j+2^i}\ a_{l+1},...,a_m$. We are interested in a suffix of the pattern whose prefix has a longest match with a prefix of the concatenated string and $a_\lambda,...$ is such a suffix.

Step 1. Construction of the suffixes tree.

The suffixes tree as it is described in section 2 has m leaves and therefore it has $< m$ internal nodes. However, the number of pairs of suffixes is $\Theta(m^2)$ which is much larger. Step 1.1 results in a set of at most $m-1$ pairs which provide all internal nodes (as implied by Observation (1) below). Observations (2) and (3) enable us to use the same computation to find two things: (a) a minimal set of pairs which provides all internal nodes, and (b) for each node, its father in the tree.

Step 1.1. Sort the m suffixes of the pattern. (A comparison between two suffixes is performed in $O(1)$ time as follows. $MAX-LENGTH$ gives the longest match of prefixes between the two suffixes. This implies the index of the leftmost character in which the two suffixes differ. So, compare the characters that have this index in both suffixes.) Any parallel sorting algorithm which is based on comparisons can be used here. We assume of course, some total order on the characters of the pattern: In the computer, each character is given by a binary representation. We can use the natural order on these binary representations.

Let $B_0, B_1, ..., B_{m-1}$ be the sorted vector of suffixes which is obtained in Step 1.1. Let f_i be the length of longest equal prefix of B_i and B_{i+1}, ($0 \le i < m-1$). Note that, the f_i values can be looked up in $MAX-LENGTH$. Step 1.2 below computes the suffixes tree from the f_i 's using the following observations.

Observations. (1) Let v be an internal node of the suffixes tree. Suppose v is defined by the longest equal prefix of two suffixes B_i and B_j. Then, there exists y, $0 \le y < m-1$, such that the longest equal prefix of B_y and B_{y+1} also defines v. (For short, we then say that v is the node of f_y). Observation (1) says that every node of the tree is a node of some f_i. This implies that it is sufficient to consider the nodes of the f_i's only for computing the suffixes tree. Observation (2) characterizes the situation where the nodes of several f_i's are the same.

(2) Suppose that for some $0 \le i < j < m-1$, $f_i = f_j$, and for all $i < y < j, f_y \ge f_i$. Then the nodes of f_i and f_j are the same. This implies that we can dispense with f_j in the minimal set of f_x values which define internal nodes. We identify the internal nodes of the suffixes tree with the f_x values that have not been dispensed with.

(3) Let $0 \le h < i < j < m-1$ and suppose $f_i > f_h$, $f_i > f_j$ and for all $h < y < j$, $f_i \le f_y$. Then, the node of the maximal among f_h and f_j is the father of the node of f_i. (Note that if $f_h = f_j$, then, by observation (2), their nodes are the same).

Step 1.2. Consider the interval of integers $[0,1,...,m-2]$. We define the singleton subintervals $[0],[1],...,[m-2]$ to be the subintervals of level 0. The $\lceil m/2 \rceil$ size-two subintervals $[0,1],[2,3],[4,5],...$ are said to be the subintervals of level 1. The $\lceil m/2^i \rceil$ size-2^i subintervals $[0,..,2^i-1]$ $[2^i,...,2.2^i-1]$,... are said to be the subintervals of level i. All these $O(m)$ subintervals are called the power-two subintervals of $[0,...,m-2]$.

1.2.1. For each power-two subinterval find the minimum f_i over the subinterval. We implement this computation using a balanced binary tree in the obvious way. This takes $O(log\ m)$ time using $m/log\ m$ processors.

1.2.2. For each $0 \le i < m-1$, find the largest $h < i$ such that $f_h \le f_i$ (if such h exists). Denote this h by $h(i)$. We implement this computation by assigning a processor to each such i. The processor finds $h(i)$ by a kind of binary search on the power-two subintervals in time $O(log\ m)$.

If $f_i > f_{h(i)}$ or if $h(i)$ does not exist, we conclude that f_i belongs to the minimal set of $f_.$ and identify f_i with a node of the suffixes tree.

If $f_i = f_{h(i)}$, we conclude that f_i does not belong to this minimal set. We say that f_i is a non-node. We compute the node $f_{\alpha(i)}$ which is the node which is defined by non-node f_i as follows.

1.2.3. For each non-node f_i we assign a processor which performs a binary search on the power-two subintervals to find the smallest $\alpha < i$ such that $f_i = f_\alpha$ and for all $\alpha < y < i, f_y \ge f_i$.

1.2.4. For each $0 \le i < m-1$, find the smallest $j > i$ such that $f_j < f_i$ (if such j exists). Denote this j by $j(i)$. We implement this computation in the same way as in 1.2.2 above. So, using $m-1$ processors it takes $O(\log m)$ time.

1.2.5. For each node of the tree, find its father.

For each internal node f_i, take the maximum value between $f_{h(i)}$ and $f_{j(i)}$. Let $\beta(i)$ be $h(i)$ if $f_{h(i)}$ provides the maximum and let it be $j(i)$ otherwise. By Observation (3), the node of $f_{\beta(i)}$ is the father of node f_i in the suffixes tree. Observe that $f_{\beta(i)}$ may be a non-node. However, in 1.2.2 above we found the node $f_{\alpha(i)}$ which is the node defined by f_i. So, using a processor per node we can find its father in $O(1)$ time.

Each suffix of the pattern is a leaf of the tree. Consider suffix B_i in the output of 1.1. Using the notations of 1.2.5 for internal nodes, $h(i)$ is f_{i-1} and $j(i)$ is f_i. Finding the father of B_i is now similar to finding the father of an internal node.

Below we use the Euler tour technique of [TV-85] and [V-85a] which computes various functions on tree. The description below has been taken from [V-85a].

Step 2. Find an Euler path in the suffixes tree that starts and ends at the root (say r) and compute for each node in the tree the length of the (shortest) path leading from the node to the root, to be called the *level* of the node. The input to Step 2 is T the suffixes-tree which was computed in Step 1.

2.1 Finding an Euler path.

> 2.1.1 Replace each edge (u,v) of T by two anti-parallel directed edges $u \to v$ and $v \to u$ to form a digraph called $\overline{T}$.
>
> 2.1.2 For each node v of T we do the following. (Let degree(v)=d in T and let the d adjacent edges of v in T be $(v,u_1),...,(v,u_d)$.)
>
> $D(u_i \to v) := v \to u_{i+1 \bmod d}$ for $1 \le i \le d$. Now D has an Euler cycle.
>
> The 'correction', $D(u_d \to r) := '\mathit{end\ of\ list}'$ (where d = degree(r)) gives an Euler path that starts and ends at r.

2.2 Finding for each node its level in T.

> 2.2.1 The distance of each edge of $\overline{T}$ from the end of the path is computed into a vector R using a 'doubling' procedure.
>
> > Initialize: $R(e) := -1$ for all edges e in $\overline{T}$ which are directed towards the root and $R(e) := 1$ for all the edge e in $\overline{T}$ which are directed away from the root. Also $R('\mathit{end\ of\ list}') := 0$.
> >
> > Apply $\lceil \log 2(m-1) \rceil$ iterations in parallel ($(m-1)$ is the length of the Euler path):
> >
> > $R(e) := R(e) + R(D(e)),\ \ D(e) := D(D(e))$.
>
> 2.2.2 The doubling procedure assigns to $R(e)$, of each edge e ($u \to v$) which is directed towards the root in $\overline{T}$, the level of u in T.

Step 3. We show how to find for each suffix of the pattern A_j and each integer i, $0 \le i \le \lceil \log m \rceil$, all values of $PAIR-FIT(j,l,i)$. So, below, j and i are fixed.

Assign a processor to each suffix A_g. Using $MAX-LENGTH$ the processor checks whether a prefix of length 2^i of A_j is a prefix of A_g. If yes, "mark" (the leaf of) suffix A_{g+2^i}. As a result some of the leaves of the tree are marked. Recall that our goal is to compute into $PAIR-FIT(j,l,i)$ (for each $l, 0 \le l \le m-1$) the values λ and f such that $a_{j+1},...,a_{j+2^i}$ $a_{l+1},...,a_{l+f} = a_{\lambda+1}, \ldots, a_{\lambda+2^i+f}$ and there is no larger f (and λ) for which such a match holds. If l is a marked leaf then we simply select $\lambda := l-2^i$ and $f := m-l$. Otherwise, *to maximize f we have to find a marked leaf such that the level of LCA(l,"the marked leaf") is as large as possible*.

Observations. Suppose leaf l is not marked. Consider the Euler path on $\overline{T}$ and the location of leaf l in this path. (The incoming edge of l is followed by the outgoing edge of l. We say that the *location* of leaf l is "between" these two edges.) Let α be the last marked leaf which precedes l in the Euler path and let β be the first marked leaf which succeeds l in the Euler path. Then,
(1) either α or β (or both) can provide the desired λ.
(2) the portion of the Euler path leading from α to l visits $LCA(\alpha,l)$; moreover, the node $LCA(\alpha,l)$ provides the minimal level in this path. Similarly, the portion of the Euler path leading from l to β visits $LCA(\beta,l)$; moreover, the node $LCA(\beta,l)$ provides the minimal level in this path.

Observation (2) explains the rest of the computation of Step 3 below:

3.1 Using a parallel prefix sum computation find for each edge in the Euler path the node of minimal level which appears between the edge and its preceding marked leaf in the Euler path. This is done as follow:

a) Divide the Euler path into subpath by throwing away all the edges from or to marked leaves.
b) Handle each subpath separately. Let each edge e $(u \rightarrow v)$ be a leaf in a balanced binary tree. Its initial value will be the level of u (computed in Step 2.2).
c) Use the parallel prefix sum computation (with one change - replace the operation sum by the operation minimum) to finish the computation.

3.2 Using a parallel prefix sum computation find for each edge in the Euler path the node of minimal level which appears between the edge and its succeeding marked leaf in the Euler path (similar to 3.1).

3.3 Determine for each l whether it preceding marked leaf or succeeding marked leaf will be selected for λ.

3.4 Using a computation which is similar to 3.1 and 3.2 above, find for each edge in the tree its preceding marked leaf and its succeeding marked leaf. (Each edge needs actually one of these data, as implied by 3.3. However, the computation is much easier if we compute both for all edges.)

3.5 Using 3.3 and 3.4 determine λ for each non marked leaf. If v is the node of minimal level between λ and l then $f := END(v)$.

Complexity. Step 3 takes $O(log\ m)$ time using $m/log\ m$ processors for each of the m values of j and $\lceil \log m \rceil + 1$ values of i. Therefore, we get $O(log\ m)$ time using m^2 processors. It is easy to achieve the same time-processor bound for Steps 1 and 2.

5.3. Computation Of Array Location

The input is the pattern $a_1, \ldots, a_m$. (We assume that the alphabet of the characters that can be used in the pattern is $1,...,\beta$, for some β.) The output is an one dimensional array $LOCATION[1,...,\beta]$. $LOCATION(i) = f$ means that character i in the alphabet appears in location f of the pattern ($a_f = i$).
The computation of the array *LOCATION* is straightforward.
Complexity. $O(log\ m)$ time using m processors. The time can be reduced to $O(1)$ if simultaneous writes by several processors into the same memory location are allowed.

Complexity of Part I. $O(logm)$ time using m^2 processors.

Acknowledgement. We are grateful to Z. Galil for encouraging us to continue improving our results.

REFERENCES

[AHU-74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[Br-74] R.P. Brent, ‘‘The parallel evaluation of general arithmetic expressions’’, JACM 21,2 (1974), 201-206.

[BM-77] R.S. Boyer and J.S. Moore, ‘‘A fast string searching algorithm’’, Comm. ACM 20 (1977), 762-772.

[CS-85] M. T. Chen and J. Seiferas, ‘‘Efficient and elegant subword tree construction’’, in A. Apostolico and Z. Galil (editors), Combinatorial Algorithms on Words, NATO ASI Series, Series F: Computer and System Sciences, Vol. 12, Springer-Verlag, 1985, 97-107.

[FL-80] M. Fisher and L. Ladner, ‘‘Parallel prefix computation’’, JACM 27,4 (1980), 831-838.

[G-84] Z. Galil, ‘‘Optimal parallel algorithms for string matching’’, Proc. 16th ACM Symposium on Theory of Computing, 1984, 240-248.

[G-85] Z. Galil, ‘‘Open problems in stringology’’, in A. Apostolico and Z. Galil (editors), Combinatorial Algorithms on Words, NATO ASI Series, Series F: Computer and System Sciences, Vol. 12, Springer-Verlag, 1985, 1-8.

[GS-83] Z. Galil and J.i. Seiferas, ‘‘Time-space-optimal string matching’’, J. Computer and System Sciences 26 (1983), 280-294.

[HT-84] D. Harel and R.E. Tarjan, ‘‘Fast algorithms for finding nearest common ancestors’’, SIAM J. Computing 13,2 (1984), 338-355.

[I-85] A.G. Ivanov ‘‘Recognition of an approximate occurrence of words on a turing machine in real time’’, Math. USSR Izvestiya, Vol. 24(1985), No. 3, 479-522.

[KMP-77] D.E. Knuth, J.H. Morris and V.r. Pratt, ‘‘Fast pattern matching in strings’’, SIAM J. Comp. 6 (1977), 322-350.

[KR-80] R.M. Karp, and M.O. Rabin, ‘‘Efficient randomized pattern-matching algorithms’’, manuscript, 1980.

[LV-85a] G.M. Landau and U. Vishkin ‘‘Efficient string matching in the presence of errors’’, Proc. 26 IEEE FOCS, 1985, 126-136. This is a preliminary version of [LV-85b] and [LV-85c].

[LV-85b] G.M. Landau and U. Vishkin ‘‘Efficient string matching with k mismatches’’, Theoret. Comput. Sci., to appear.

[LV-85c] G.M. Landau and U. Vishkin ‘‘Efficient string matching with k differences’’, TR-36/85, Department of Computer Science, Tel Aviv University, 1985.

[LVN-85] G.M. Landau, U. Vishkin and R. Nussinov ‘‘An efficient string matching algorithm with k differences for nucleotide and amino acid sequences ’’, Nucleic Acids Research 1986, to appear.

[ML-84] M.G. Main and R.J. Lorentz, ‘‘An $O(n \log n)$ algorithm for finding all repetitions in a string’’, J. of Algorithms (1984), pp. 422-432.

[S-80] P. H. Sellers, ‘‘The theory and computation of evolutionary distances: Pattern recognition’’, J. of Algorithms 1 (1980), 359-373.

[SK-83] D. Sankoff and J.b. Kruskal (editors), Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison, Addison-Wesley, Reading, MA, 1983.

[TV-85] R.E. Tarjan and U. Vishkin, ‘‘An efficient parallel biconnectivity algorithm’’, SIAM J. Comput. 14,4(1985),862-874.

[U-83] E. Ukkonen, ‘‘On approximate string matching’’, Proc. Int. Conf. Found. Comp. Theor., Lecture Notes in Computer Science 158, Springer-Verlag, 1983, 487-495.

[U-85] E. Ukkonen, ‘‘Finding approximate pattern in strings’’, J. of Algorithms 6 (1985), 132-137.

[V-83] U. Vishkin, ‘‘Synchronous parallel computation - a survey’’, TR-71, Dept. of Computer Science, Courant Institute, NYU, 1983.

[V-84] U. Vishkin, ‘‘An optimal parallel connectivity algorithm’’, Discrete Applied Math. 9 (1984), 197-207.

[V-85a] U. Vishkin, ‘‘On efficient parallel strong orientation’’, Information Processing Letters 20 (1985), 235-240.

[V-85b] U. Vishkin, ‘‘Optimal parallel pattern matching in strings’’, Proc. 12th ICALP, Lecture Notes in Computer Science 194, Springer-Verlag, 1985, 497-508. Also, Information and Control, to appear.

[W-73] P. Weiner ‘‘Linear Pattern Matching Algorithm’’, Proc. 14 IEEE Symposium on Switching and Automata Theory, 1973, 1-11.

[WF-74] R. Wagner and M. Fisher ‘‘The string-to-string correction problem’’, J. ACM 21 (1974), 168-178.