Algorithmica (1988) 3: 347–365

Algorithmica
© 1988 Springer-Verlag New York Inc.

Parallel Construction of a Suffix Tree with Applications[1]

A. Apostolico,[2] C. Iliopoulos,[2] G. M. Landau,[3] B. Schieber,[3] and U. Vishkin[3,4]

Abstract. Many string manipulations can be performed efficiently on suffix trees. In this paper a CRCW parallel RAM algorithm is presented that constructs the suffix tree associated with a string of n symbols in $O(\log n)$ time with n processors. The algorithm requires $\Theta(n^2)$ space. However, the space needed can be reduced to $O(n^{1+\varepsilon})$ for any $0 < \varepsilon \leq 1$, with a corresponding slow-down proportional to $1/\varepsilon$. Efficient parallel procedures are also given for some string problems that can be solved with suffix trees.

Key Words. Parallel algorithms, CRCW RAM, Suffix trees, On-line string matching, Longest repeated substring in a string, Approximate string matching, Skeleton trees, Processor allocation techniques.

1. Introduction. Let $x = x_1, x_2, \ldots, x_n$ be a string of $n = |x|$ *symbols* and assume that x_n is a special symbol # that occurs nowhere else in x. We use I to denote the *alphabet* of x, i.e., the set of all distinct symbols occurring in x. (Note that $|I| \leq n$.) Given a substring w of x, a *descriptor* of w is any pair $(i, |w|)$ such that i is the starting position in x of an occurrence of w. The *suffix tree* T_x associated with x is the trie (digital search tree) with n leaves and at most $n-1$ internal nodes such that:

(1) Each edge is labeled with a descriptor of some substring of x.
(2) No two sibling edges may have the same (nonempty) prefix.
(3) Each leaf is labeled with a distinct position of x.
(4) The concatenation of the labels on the path from the root to leaf i describe the suffix of x starting at position i.

(See Figure 1 for an example.) In practice, the label of the edge connecting node μ to its parent node is stored in μ. Observe that, in general, there is more than one way to assign consistent labels to the edges of a suffix tree.

[1] The results of this paper have been achieved independently and simultaneously in [AI-86] and [LSV-86]. The research of U. Vishkin was supported by NSF Grant NSF-CCR-8615337, ONR Grant N00014-85-K-0046, and Foundation for Research in Electronics, Computers, and Communication, administered by the Israeli Academy of Sciences and Humanities. The research of A. Apostolico was carried out in part while visiting at the Istituto di Analisi dei Sistemi e Informatica, Rome, with support from the Italian National Research Council. The research of G. M. Landau, B. Schieber, and U. Vishkin was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC02-76ER03077.
[2] Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA.
[3] Department of Computer Science, School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel 69978.
[4] Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, USA.

Received November 3, 1986; revised November 30, 1987. Communicated by Jeffrey Scott Vitter.

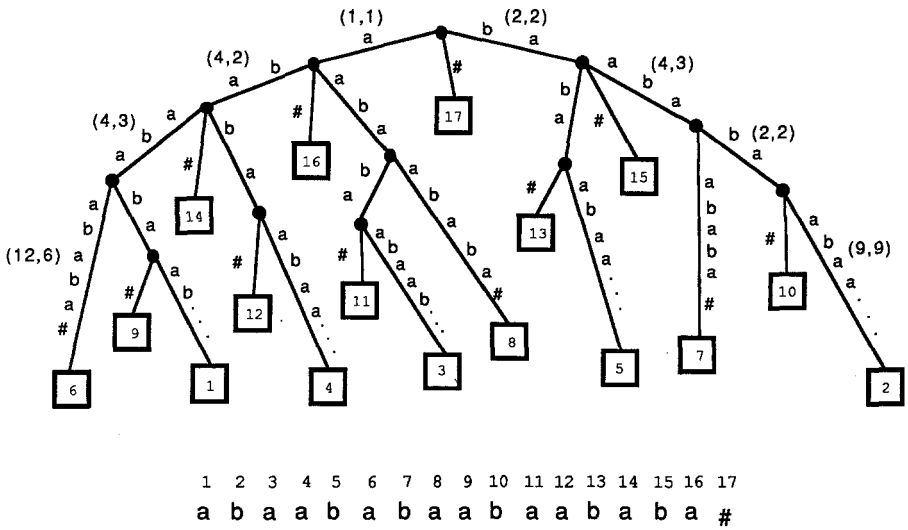

Fig. 1. The suffix tree T_x for $x = abaababaabaababa\#$. For clarity, the arcs of T_x are labeled by substrings of x rather than by substring descriptors. Such descriptors are given on the two outermost paths of the tree, as a sample illustration.

The main problem addressed in this paper is the parallel construction of the suffix tree T_x associated with input string x. For fixed alphabet size, the sequential algorithms in [We-73] and [Mc-76] construct T_x in linear time. The time bound becomes $O(n \log|I|)$ if the alphabet size is not a constant. Suffix trees and their companion structures support many string manipulations, such as performing on-line string matching [AHU-74], finding the longest repeated substring in a string, testing square-freedom of a string [AP-83], [Ap-84], finding all the squares or repetitions in a string [AP-83], computing substring statistics with or without overlap [AP-85a], [AP-85b], and performing exact [AG-86] or approximate [LV-86] pattern matching. A more detailed list of applications is given in [Ap-85]. In the context of parallel computation, various open problems revolve around T_x [Ga-85]. The only previous parallel algorithm for constructing suffix trees is given in [LV-86]. It runs in time $O(\log n)$ and uses $n^2/\log n$ processors.

We adopt the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM) model of computation. We use n processors which can simultaneously read from and write to a common memory with $\Theta(n^2)$ locations. In case several processors seek access to the same memory location for write purposes, one of them succeeds but we do not know in advance which. See [Vi-83] for a survey of results concerning PRAMs. The overall *processors* × *time* cost of our algorithm is $O(n \log n)$, which is optimal when $\log|I|$ is of the same order of magnitude as $\log n$. Although the algorithm requires quadratic space, only $O(n \log n)$ locations need initialization. Moreover, we show later that the space can be reduced to $O(n^{1+\varepsilon})$, for any chosen $0 < \varepsilon \leq 1$, with a corresponding slow-down proportional to $1/\varepsilon$.

Our approach to the construction of T_x consists of two main parts. In the first part, described in Section 2, an approximate version of the tree is built, called the *skeleton*. This part of the construction is reminiscent of an early approach to subquadratic pattern matching [KMR-72]. The second part, described in Section 3, consists of refining the skeleton to transform it into T_x. The processor allocation technique that is used for the refinement is of independent interest. Allocating processors to jobs is often a crucial task in the design of efficient parallel algorithms, and there are papers mainly devoted to overcoming allocation problems. For example, [SV-81] solved the allocation problem in the algorithm of [Va-75] for finding the maximum among n elements, [BH-83] and [Kr-83] solved the allocation problem in the algorithm of [Va-75] for merging. [CV-86a], [CV-86b], and [Vi-84] gave deterministic and randomized allocation schemes for list ranking.

Section 4 contains a brief analysis of the various allocation techniques that can be used for a suffix tree. In Section 5 we show how the space used in our construction can be reduced. Finally, we describe in Section 6 how our suffix tree construction leads to the design of efficient parallel algorithms for on-line string matching, finding a longest repeated substring in a string, and performing approximate pattern matching.

2. Constructing the Skeleton Tree. From now on we will assume without loss of generality that n is a power of 2. We also extend x by appending to it $n-1$ instances of the symbol $\#$. We use $x\#$ to refer to this modified string. We now list some salient features of the skeleton tree D_x of x, and then give a constructive definition of D_x. The basic structure of the skeleton for the string of Figure 1 is shown in Figure 2. The skeleton D_x of x is a tree with n leaves. Each internal node of D_x has at least two children. The edges in D_x point from each node to its parent. Each leaf or internal node of D_x is labeled with the descriptor of some substring of $x\#$ having starting positions in $[1, n]$. If node μ is labeled with descriptor (i, l), then $l = 2^q$ for some q, $0 \leq q \leq \log n$. If μ is a leaf then $l = n$. If μ is an internal node other than the root, then q is the *stagenumber* of μ. If the label of μ corresponds to substring w of x, then we write $w = W(\mu)$, and we call μ the *locus* of w. A constructive definition for D_x is as follows:

(i) The root of D_x is the locus of the empty word. The root has $|I|$ sons, each one being the locus of a distinct symbol of I.

(ii) Assume that all nodes of stagenumber up to $l-1 \geq 0$ have been inserted in D_x. To expand D_x to stagenumber $l \leq \log n$, consider the nodes of stagenumber $l-1$ one by one. For a generic such node μ, let $w = W(\mu)$. Now do the following:

1. If $w = z\#$ for some string z over I, then make μ the (unique) leaf labeled (i, n), where i is the first component of the old label of μ.
2. Assume instead that w cannot be written as $z\#$ for some string z over I. Let $\{s_1, s_2, \ldots, s_k\}$ be a set of maximum cardinality among the sets formed by distinct substrings of $x\#$ with the properties: $|s_t| = 2|w|$ and w is a prefix of s_t, $t = 1, 2, \ldots, k$. (Thus, if i is the starting position of an

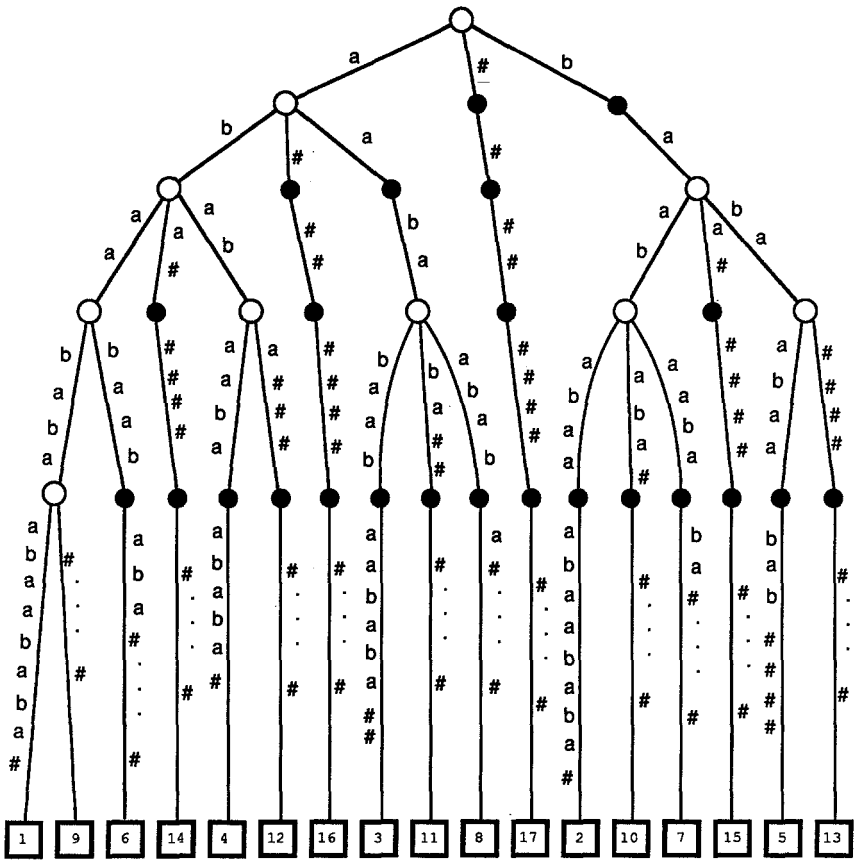

Fig. 2. Basic structure of the skeleton tree D_x for the string of Figure 1. Solid points are used to mark nonbranching nodes. Such nodes are introduced while constructing D_x, but they are also removed during the construction. Node labels are not reported in the figure.

occurrence of w in $x\#$, then there is some s_t also starting at i. In the string of Figure 1, for example, we have that each occurrence of $w = ab$ in $x\#$ extends into either $s_1 = abaa$, or $s_2 = aba\#$, or $s_3 = abab$. On the other hand, $w = aa$ occurs in $x\#$ only as a prefix of $s_1 = aaba$. Note that, in general, an s_t may occur more than once in $x\#$.) We distinguish two cases. (a) $k > 1$. We create k sons of μ, $v_1, v_2, \ldots, v_k$, and make v_t the locus of s_t, $t = 1, 2, \ldots, k$. (b) $k = 1$, i.e., w occurs always as a prefix of the same substring s_1. We make μ the locus of s_1.

Observe that no two nodes of D_x can have the same label. A natural parallel construction of D_x is based on the above definition. We describe such a construction in detail, to acquaint the reader with the basic concurrent steps which are used throughout this paper.

We use n processors $p_1, p_2, \ldots, p_n$, where i is the *serial number* of processor p_i. At the beginning, processor p_i is assigned to the ith position of x, $i = 1, 2, \ldots, n$.

It is convenient to think of each processor as being assigned two segments of the common memory, each segment consisting of $\log n + 1$ cells. The segments assigned to p_i are called ID_i and $NODE_i$, respectively. By the end of the computation, $ID_i[q]$ $(i = 1, 2, \ldots, n;\ q = 0, 1, \ldots, \log n)$ contains (the first component of) a descriptor for the substring of $x\#$ of length 2^q which starts at position i in $x\#$, with the constraint that all the occurrences of the same substring of x get the same descriptor. If, for some value of $q < \log n$, $NODE_i[q]$ is not empty, then it represents a node μ of stagenumber q in D_x, as follows: the field $NODE_i[q].LABEL$ is a replica of $ID_i[q]$, and the field $NODE_i[q].PARENT$ points to the location of the parent of μ. Finally, $NODE_i[\log n]$ stores the leaf labeled (i, n) and thus is nonempty for $i = 1, 2, \ldots, n$. For convenience, we extend the notion of ID to all positions $i > n$ through the convention: $ID_i[q] = n + 1$ for $i > n$. The computation makes crucial use of a *bulletin board* (BB) of $n \times (n+1)$ locations in the common memory. All processors can simultaneously write to BB and simultaneously read from it. We use the following concurrent-write convention. In case several processors try simultaneously to write into the same memory location, one of them succeeds but we do not know in advance which. In the following we call $winner(i)$ the index of the processor which succeeds in writing to the location of the common memory attempted by p_i.

Procedure *Skeleton-Tree* takes as input the string x and a location of the common memory called $ROOT$, and computes the entries of the arrays $NODE_i[q]$, $ID_i[q]$ $(i = 1, 2, \ldots, n,\ q = 0, 1, \ldots, \log n)$. The procedure consists of some initializations, that implement point (i) in the definition of D_x, and $\log n$ main iterations, implementing point (ii).

The initializations are as follows. In parallel, all processors initialize their $NODE$ and ID arrays. Next, processors facing the same symbol of I attempt to write their serial number in the same location of BB. Say, if $x_i = s \in I$, processor p_i attempts to write i in $BB[1, s]$. Through a second reading from the same location, p_i reads $j = winner(i)$ and sets $ID_i[0] \leftarrow j$. (Thus $(j, 1)$ becomes the descriptor for every occurrence of symbol s.) For all i such that $winner(i) = i$, processor p_i sets $NODE_i[0].PARENT \leftarrow address(ROOT)$ and copies $ID_i[0] = i$ into $NODE_i[0].LABEL$. Hence $NODE_i[0]$ becomes the locus of s.

We now describe *iteration* q, $q = 0, 1, \ldots, \log n - 1$, which is also performed synchronously by all processors. First, processor p_i, $i = 1, 2, \ldots, n$, creates a composite label TID_i, by setting: $TID_i \leftarrow (ID_i[q],\ ID_{i+2^q}[q])$. Next, processor p_i attempts to write i in $BB[TID_i] = BB[ID_i[q],\ ID_{i+2^q}[q]]$. Now, processor p_i sets: $ID_i[q+1] \leftarrow winner(i)$, $i = 1, 2, \ldots, n$. The processors that are not winners become idle for the remainder of the stage. On the other hand, any winner p_i performs the following:

$NODE_i[q+1].PARENT \leftarrow (ID_i[q],\ q)$
$NODE_i[q+1].LABEL \leftarrow ID_i[q+1]$
if $NODE_{ID_i[q]}[q]$ has only one child **then**
  begin
    $NODE_i[q+1].PARENT \leftarrow NODE_{ID_i[q]}[q].PARENT$;
    Make $NODE_{ID_i[q]}[q]$ empty.
  end

Thus, the winners create new locuses in their associated *NODE* locations. Whenever a node μ is created that has no siblings, then the pointer from $parent(\mu)$ is removed and copied into μ. This avoids the formation of chains of unary nodes.

The existence of siblings can be checked as follows. Assume that for each row r of *BB*, there is a distinct memory location, say $AUX[r]$, known to all processors. At each stage, there are siblings iff two or more successful processors write to different locations of the same row of *BB*. To find out whether this is the case, all successful processors writing in the same row r of *BB* attempt to write their index in $AUX[r]$. Next, all the processors in that row except the winner write a special marker in $AUX[r]$. Finally, all the processors in the same row check the status of $AUX[r]$. Clearly, processor p_i was the only successful processor in row r iff, at the time of checking, $AUX[r] = i$.

The correctness of the procedure follows by straightforward induction. Since no two n-symbol substrings of $x\#$ are identical, processor p_i $(i = 1, 2, \ldots, n)$ must be occupying the "leaf" $NODE_i[\log n]$ at the end of the computation. The time complexity is obviously $O(\log n)$. Note that $NODE_i[q].LABEL$ not empty implies $NODE_i[q].LABEL = (i, 2^q)$, that is, the label of a node, when defined, is nothing but the address of that node. Although the *LABEL* fields are entirely redundant so far, assuming this node format from the start simplifies the rest of our presentation. Finally, we remark that *BB* need not be initialized.

3. Refining D_x. By the end of the construction of D_x, processor p_i will be occupying leaf i, $i = 1, 2, \ldots, n$. Prior to starting the transformation of D_x into T_x, the labels of all nodes of D_x have to be modified as follows. Recall that the current *LABEL* of a node μ is a starting position of $W(\mu)$ in $x\#$ which is also the address of this node. The modified label (*m-label*) to be constructed for μ is any pair (i, l) such that, letting $W(\mu) = W(parent(\mu)) \cdot w$, it is $l = |w|$ and i is the starting position of an occurrence of w in $x\#$. In the following, we call the m-labeled skeleton the tree that is obtained by substituting every label of D_x with a consistent m-label. Setting aside the orientation of edges, the main difference between T_x and the m-labeled skeleton D_x is that in T_x there cannot be two sibling nodes such that their labels describe two substrings of x having a common prefix (i.e., D_x is not a trie). However, the m-labeled D_x shares with T_x the properties (1), (3), and (4) listed in defining the latter, provided $x\#$ is used there in the place of x.

A processor can trivially compute the m-label of μ in constant time knowing the *LABEL* of μ, and the stagenumbers, say q and q', of μ and $parent(\mu)$, respectively. Formally, if j is the *LABEL* of μ, then $(j + 2^q, 2^q - 2^{q'})$ is the m-label of μ. The n processors can produce all m-labels in $\log n$ parallel steps. Using the parent pointers, the processors migrate toward *ROOT* with a synchronous pace based on stagenumbers: the m-labels of all children of nodes with the same stagenumber are computed at the same time. (Recall that the difference in stagenumber between a node and its parent is not necessarily 1.) At the beginning, all processors occupying leaves which are children of nodes of stagenumber $\log n - 1$ change the labels of these nodes into m-labels. Next, the processors

compete for the common parent node, say, by attempting to simultaneously write on it the labels (addresses) of the nodes which they currently occupy. The winners are marked "free": they ascend to the parent node where they will perform the necessary label adjustment at the appropriate stage. The losers simply take a record of the (old) label used by the winner. The $(q-1)$th iteration involves all free processors on nodes with a stagenumber of q or higher. The operation is the same as above.

A by-product of the m-label construction process is a mapping that assigns some leaves and internal nodes to processors in such a way that the following property is met.

PROPERTY 1. If a node other than *ROOT* has k children, then precisely $k-1$ of the children have been assigned a processor. Moreover, each one of the $k-1$ processors knows the address of the unique sibling without a processor.

The proof of Property 1 is straightforward. Let now (i, l) and (j, m) be the m-labels of two sibling nodes μ and ν of D_x, and let q be the stagenumber of $parent(\mu) = parent(\nu)$.

FACT 1. The substrings of $x\#$ whose descriptors are the m-labels of μ and ν have a common prefix of length at most $2^q - 1$.

FACT 2. If k is the length of the longest common prefix of $x\#[i, i+l-1]$ and $x\#[j, j+m-1]$, then $ID_i[\lfloor \log k \rfloor] = ID_j[\lfloor \log k \rfloor]$.

Fact 1 follows from the definition of D_x, Fact 2 holds by the construction of the *ID*s.

Assuming a fixed-size alphabet, the transformation of the m-labeled D_x into T_x is carried out in two steps. First, a tree is produced that is identical to T_x save the fact that all edges are directed upward, as in D_x. Next, the directions of all edges are reversed.

The first and more important step is actuated by producing $\log n - 1$ consecutive *refinements* of $D_x = D^{(\log n - 1)}$. The qth such refinement is denoted by $D^{(\log n - q - 1)}$. Informally, $D^{(\log n - q - 1)}$ is a labeled tree with n leaves and no unary nodes which has much the same structure of the m-labeled D_x. In particular, properties (1), (3), and (4) of the definition of T_x hold for any refinement of D_x. The refinement $D^{(0)}$ is identical to T_x except for the edge directions. Figure 3 shows the second refinement for our example skeleton.

We now give rigorous definitions for $D^{(\log n - q - 1)}$, $q = 1, 2, \ldots, \log n - 1$. We do so by specifying how $D^{(\log n - q - 1)}$ is obtained from $D^{(\log n - (q-1) - 1)}$, for $q = 1, 2, \ldots, \log n - 1$. For simplicity, we use k henceforth to denote $\log n - q - 1$. First, two more definitions are needed. A *nest* is any set formed by all children of some node in $D^{(k)}$. Let (i, l) and (j, k) be the labels of two nodes in some nest of $D^{(k)}$. An integer t, $0 < t \leq \min[l, k]$, is a *refiner* for (i, l) and (j, k) iff $x\#[i, i+t-1] = x\#[j, j+t-1]$.

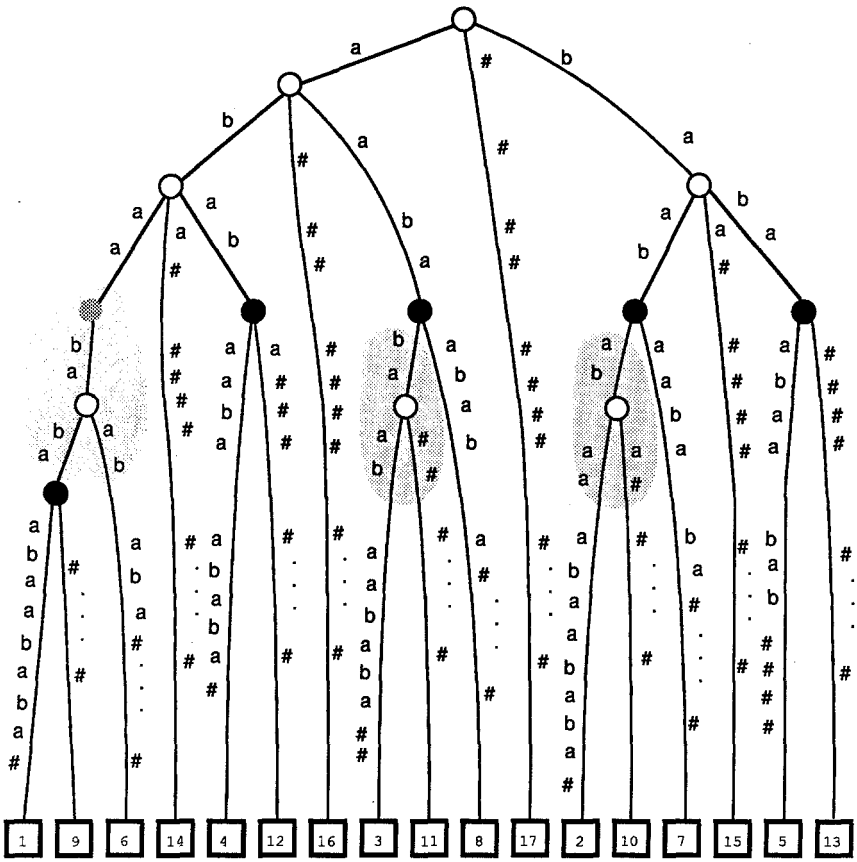

Fig. 3. No nest of the skeleton of Figure 2 undergoes changes in the first refining stage. The effect of the second refining stage is visible in the lightly shaded areas of the present figure. Parent nodes of the nests that were eligible at the inception of this stage are shown solid. Among the effects of this stage, the old locus of *abaa* (shown shaded) is eliminated from the tree. One more refining stage leads to the tree of Figure 1.

Assume now that all refinements down to $D^{(k)}$, $\log n - 1 \leq k < 0$, have already been produced, and that $D^{(k)}$ meets the following *condition*(k):

(i) $D^{(k)}$ is a labeled tree with n leaves and no unary nodes.
(ii) $D^{(k)}$ enjoys properties (1), (3), and (4) of the definition of T_x.
(iii) $D^{(k)}$ is labeled in such a way that no pair of labels of nodes in the same nest admits a refiner of size 2^k.

Observe that *condition*$(\log n - 1)$ is met trivially by D_x. Moreover, part (iii) of *condition*(0) implies that reversing the direction of all edges of $D^{(0)}$ would change $D^{(0)}$ into a digital-search tree that stores the collection of all suffixes of x. Clearly, such a trie fulfills precisely the definition of T_x.

We now define $D^{(k-1)}$ as the tree obtained by transforming $D^{(k)}$ as follows. The manipulations that transform $D^{(k)}$ into $D^{(k-1)}$ are performed synchronously on all and only the *eligible* nests of $D^{(k)}$, i.e., on those nests that might admit a refiner of size $2^{(k-1)}$. Clearly, the only eligible nests in D_x are those whose parent nodes have stagenumber $\log n - 1$. There is only one such nest in the skeleton of Figure 2, namely, that formed by leaves 1 and 9 (however, this nest does not have a refiner of size $2^{(\log n-1)-1} = n/4$). The nests of nodes whose parents have stagenumber $\log n - 2$ become eligible at the inception of the second refining stage (see Figure 3), and so on.

Assume that, in $D^{(k)}$, all nodes that are parents of currently eligible nests are suitably marked. Let $(i_1, l_1), (i_2, l_2), \ldots, (i_m, l_m)$ be the set of all labels in some eligible nest of $D^{(k)}$. Let v be the parent node of that nest. The nest is refined in two steps.

Step 1. Use the *LABEL* and *ID* tables to modify the nest rooted at v as follows. With the child node labeled (i_j, l_j) associate the *split*-label $(ID_{i_j}[k-1], ID_{i_j+2^{k-1}}[k-1])$, $j = 1, 2, \ldots, m$. Now partition the children of v into equivalence classes, putting in the same class all nodes with the same first component of their split-labels. For each nonsingleton class which results, perform the following three operations:

(1) Create a new parent node μ for the nodes in that class, and make μ a son of v.
(2) Set the *LABEL* of μ to $(i, 2^{(k-1)})$, where i is the first component of the split-label of all nodes in the class.
(3) Consider each child of μ. For the child whose current *LABEL* is (i_j, l_j), change *LABEL* to $(i_j + 2^{(k-1)}, l_j - 2^{(k-1)})$.

Step 2. If more than one class resulted from the partition, then stop. Otherwise, let C be the unique class resulting from the partition. It follows from assumption (i) on $D^{(k)}$ that C cannot be a singleton class. Thus a new parent node μ as above was created for the nodes in C during step 1. Make μ a child of the parent of v and set the *LABEL* of μ to $(i, l + 2^{(k-1)})$, where (i, l) is the label of v.

The following theorem shows that our definition of the series of refinements $D^{(k)}$ is unambiguous.

THEOREM 1. *The synchronous application of steps* 1 *and* 2 *to all eligible nests of* $D^{(k)}$ *produces a tree that meets condition*$(k-1)$.

PROOF. Properties (ii) and (iii) of *condition*$(k-1)$ are easily established for $D^{(k-1)}$. Thus we concentrate on property (i). Since no new leaves were inserted in the transition from D^k to $D^{(k-1)}$, property (i) will hold once we prove that $D^{(k-1)}$ is a tree with no unary nodes.

Clearly, the nest of the children of the root is not eligible for any $k > 0$. Thus, for any parent node v of an eligible nest of $D^{(k)}$, *parent*(v) is defined. By

condition(k), node v has more than one child, and so does *parent*(v). Let $\bar{D}^{(k)}$ be the structure resulting from application of step 1 to $D^{(k)}$.

If, in $D^{(k)}$, the nest of *parent*(v) is not eligible, then v is a node of $D^{(k-1)}$, and v may be the only unary node in $\bar{D}^{(k)}$ between any child of v in $D^{(k)}$ and the parent of v in $D^{(k)}$. Node v is removed in step 2, unless v is a branching node in $\bar{D}^{(k)}$. Hence no unary nodes result in this part of $D^{(k-1)}$.

Assume now that, in $D^{(k)}$, both the nest of v and that of *parent*(v) are eligible. We claim that, in $\bar{D}^{(k)}$, either the parent of v has not changed and it is a branching node, or it has changed but is still a branching node. Indeed, by definition of $D^{(k)}$, neither the nest of v nor that of *parent*(v) can be refined into only one singleton equivalence class. Thus, by the end of step 1, the following alternatives are left:

1. The parent of v in $\bar{D}^{(k)}$ is identical to *parent*(v) in $D^{(k)}$. Since the nest of *parent*(v) could not have been refined into only one singleton class, then *parent*(v) must be a branching node in $D^{(k-1)}$. Thus this case reduces to that where the nest of *parent*(v) is not eligible.
2. The parent of v in $\bar{D}^{(k)}$ is not the parent of v in $D^{(k)}$. Then *parent*(v) in $\bar{D}^{(k)}$ is a branching node, and also a node of $D^{(k-1)}$. If v is a branching node in $\bar{D}^{(k)}$, then there is no unary node between v and *parent*(v) in $\bar{D}^{(k)}$, and the same holds true between any node in the nest of v and v. If v is a unary node in $\bar{D}^{(k)}$, then the unique child of v is a branching node. Since the current parent of v is also a branching node by hypothesis, then removing v in step 2 eliminates the only unary node existing on the path from any node in the nest of v to the closest branching ancestor of that node. □

In order to specify which nests of $D^{(k-1)}$ are eligible, we need to complete the rules for eligibility. In the light of the preceding discussion, it is easy to see that, once a node has become the parent of an eligible nest, it will not lose this property through the subsequent refinements (as long as it is not eliminated from the tree), even though the nest itself may undergo changes. Moreover, the nests of nodes created in producing $D^{(k-1)}$ are eligible for the transition from $D^{(k-1)}$ to $D^{(k-2)}$.

If the nest of $D^{(k)}$ rooted at v had a row R of BB all to itself, then the transformation undergone by this nest in step 1 can be accomplished by m processors in constant time, m being the number of children. Each processor handles one child node. It generates the split-label for that node using its *LABEL* and the *ID* tables. Next, the processors use the row of BB assigned to the nest and the split-labels to partition themselves into equivalence classes: each processor in the nest whose split-label has first component i competes to write the address of its node in the ith location of R. A *representative* processor is elected for each class in this way. Singleton classes can be trivially spotted through a second concurrent write restricted to losing processors (after this second write, a representative processor which still reads its node address in R knows itself to be in a singleton class). The representatives of each nonsingleton class now create the new parent nodes, label them with the first component of their split-label, and make each new node accessible by all other processors in the class. To conclude step 1, the processors in the same class update the labels of their nodes.

For step 2, the existence of more than one equivalence class needs to be tested. This is done through a competition of the representatives which uses the root of the nest as a common write location, and follows the same mechanism as in the construction of D_x. If only one equivalence class was produced in step 1, then its representative performs the adjustment of the label prescribed by step 2.

The above discussion suggests that, once each node of, say, $D_x = D^{(\log n-1)}$ is assigned to a distinct processor, $D^{(\log n-2)}$ could be produced in constant time. The difficulty, however, is how to assign the nodes (notably, the newly inserted ones) of $D^{(\log n-2)}$ in constant time. It turns out that bringing fewer processors into the game leads to a crisp (re-)assignment strategy.

By definition, $D^{(k)}$ does not have unary nodes. It is seen then that the manipulations of steps 1 and 2 can be operated in constant time by assigning $m-1$ processors, rather than m to a nest of m nodes. The only additional assumption to be made is that, at the beginning, all $m-1$ processors have access to the unique node which lacks a processor of its own. Before starting step 1, the processors elect one of them to serve as a substitute for the missing processor. After each elementary step, this simulator "catches-up" with the others.

In view of Property 1, this shows that n processors can achieve the first refinement of D_x. As to the assignment of the rows of BB to the nodes of $D^{(k)}$, simply assign the ith row to processor p_i. Then, whenever p_i is in charge of the simulation of the missing processor in a nest, its BB row is used by all processors in that nest.

For any given value of k, let a *legal* assignment of processors to the nodes of $D^{(k)}$ be an assignment that enjoys Property 1.

THEOREM 2. *Given a legal assignment of processors for $D^{(k)}$, a legal assignment of processors for $D^{(k-1)}$ can be produced in constant time.*

PROOF. We give first a constant-time policy that reallocates the processors in each nest of $D^{(k)}$ on the nodes of $\bar{D}^{(k)}$. We then show that our policy leads to a legal assignment for $D^{(k-1)}$.

Let ν be the parent of a nest of $D^{(k)}$. A node to which a processor has been assigned is called *pebbled.* By hypothesis, all but one of the children of ν are pebbled. Also, all children of ν are nodes of $\bar{D}^{(k)}$. In the general case, some of the children of ν in $D^{(k)}$ are still children of ν in $\bar{D}^{(k)}$, while others became children of newly inserted nodes $\mu_1, \mu_2, \ldots, \mu_t$. Our policy is as follows. At the end of step 1, for each node μ_r of $\bar{D}^{(k)}$ such that all children of μ_r are pebbled, one pebble (say, the representative processor) is chosen among the children and passed on to the parent. In step 2, whenever a pebbled node ν is removed, then its pebble is passed down to the (unique) son μ of ν in $\bar{D}^{(k)}$.

Clearly, our policy can be implemented in constant time. To prove its correctness, we need to show that it generates a legal assignment for $D^{(k-1)}$. It is easy to see that if node ν is removed in the transition from $\bar{D}^{(k)}$ to $D^{(k-1)}$, then the unique son μ of ν in $\bar{D}^{(k)}$ is unpebbled in $\bar{D}^{(k)}$. Thus, in step 2, it can never happen that two pebbles are moved onto the same node of $D^{(k-1)}$.

By definition of $D^{(k)}$, the nest of node v cannot give rise to a singleton class. Thus at the end of step 1, either (Case 1) the nest has been refined in only one (nonsingleton) class, or (Case 2) it has been refined in more than one class, some of which are possibly singleton classes. Before analyzing these two cases, define a mapping f from the children in the nest of the generic node v of $D^{(k)}$ into nodes of $D^{(k-1)}$ as follows. If node μ is in the nest of v and also in $D^{(k-1)}$ then set $\mu' = f(\mu) = \mu$; if instead μ is not in $D^{(k-1)}$, let $\mu' = f(\mu)$ be the (unique) son of μ in $\bar{D}^{(k)}$.

In Case 1, exactly one node μ is unpebbled in $\bar{D}^{(k)}$. All the nodes μ''s are siblings in $D^{(k-1)}$ and, by our policy, μ' is pebbled in $D^{(k-1)}$ iff μ is pebbled in $D^{(k)}$.

In Case 2, node v is in $D^{(k-1)}$. Any node μ in the nest of v is in $\bar{D}^{(k)}$. At the end of step 2, the pebble of node μ will go untouched unless μ is in a nonsingleton equivalence class. Each such class generates a new parent node, and a class passes a pebble on to that node only if all the nodes in the class were pebbled. Thus, in $D^{(k-1)}$, all the children of v except one are pebbled by the end of step 1. Moreover, for each nonsingleton equivalence class, all nodes in that class but one are pebbled. At the end of step 2, for each node μ which was in the nest of v in $D^{(k)}$, node μ' is pebbled iff μ was pebbled at the end of step 1, which concludes the proof. □

4. Storing a Suffix Tree. In some advanced applications (see, for example, [AP-83], [AP-85a], [AP-85b], and [LV-86]), T_x needs only to be traversed bottom-up. The structure achieved for $D^{(0)}$ would suffice for thise tasks. Like any trie, however, T_x is usually employed to perform downward searches, starting at its root. This requires the insertion, for each original directed edge (μ, v) of $D^{(0)}$, of a *matched* downward edge (v, μ). Correspondingly, each node v must now store appropriate *downward* labels for all the downward edges originating from it. Such labels supply the branching information needed in the course of a downward search in T_x of a string w. We examine two different ways of defining such information. More precisely, let (i, l) be the label of the upward edge (μ, v). One way is to label the matched downward edge (v, μ) with the symbol of I that corresponds to x_i. This entails that the branching decision at each node be driven by the symbol that occupies a certain position of w. The second way is to use the value of $ID_i[0]$. To use this information during a search, an auxiliary table must have been precomputed that maps each symbol of I into its corresponding ID.

In either case, the set of downward labels of each internal node of T_x can be stored using a linear list, a binary trie, or an array. Resorting to arrays enables searching for w in T_x in time $O(|w|)$, but requires space $\Theta(|I| \cdot n)$ or $\Theta(n^2)$ (depending on the labeling convention adopted) to store T_x. Lists or binary tries require only linear space for T_x. However, the best time bounds for searching w under the two labeling conventions become $O(|w| \log |I|)$ and $O(|w| \log n)$, respectively. Such bounds refer to the implementation with binary tries. For ordered alphabets, the bound $O(|w| \log |I|)$ extends also to the list implementation of the

symbol-based downward labels. We describe below the trie implementation of symbol-labels and the array implementation of *ID*-labels, since all the others can be derived from one of these two quite easily.

We show how to implement symbol-based downward labels with tries, i.e., how to replace each original internal node of $D^{(0)}$ with a binary trie indexing to a suitable subset of I. This transformation can be obtained in $O(\log|I|)$ time using the legal assignment of processors that holds on $D^{(0)}$ at completion. We outline the basic mechanism and leave the details as an exercise. We simply perform $\log|I|$ further refinements of $D^{(0)}$, for which the *ID* tables are not needed. In fact, the best descriptor for a string of $\log|I|$ bits or less is the string itself. Thus, we let the processors in each nest partition their associated nodes into finer and finer equivalence classes, based on the bit-by-bit inspection of their respective symbols. Clearly, a processor occupying a node with upward label (i, l) will use symbol x_i in this process. Whenever a new branching node v is created, one of the processors in the current nest of v climbs to $\mu = father(v)$ and assigns the appropriate downward label to μ. At the end, the processors assign downward labels to the ultimate fathers of the nodes in the nest.

Finally, we discuss the array implementation of *ID*-based downward labels. This representation is needed in Section 6. We assign a vector of size n, called OUT_v, to each node v of $D^{(0)}$. The vector OUT_v stores the downward edges from v as follows. If μ is a son of v and the upward label of μ is (i, l), a pointer to μ is stored in $OUT_v[ID_i[0]]$. It is an easy exercise to show that n processors legally assigned to $D^{(0)}$, and equipped with $\Theta(n)$ locations each, can construct this implementation of T_x in constant time. In fact, the same can be done with any $D^{(k)}$, but the space needed to accommodate *OUT* vectors for all refinements $D^{(k)}$ would become $\Theta(n^2 \log n)$. Observe that, since n processors cannot initialize $\Theta(n^2)$ spaces in $O(\log n)$ time, the final collection of *OUT* vectors will describe in general a graph containing T_x plus some garbage. T_x can be separated from the rest by letting the processors in each nest convert the *OUT* vector of the parent node into a linked list. This task is accomplished trivially in extra $O(\log n)$ time. The interested reader may refer to [FL-80]. For one of the applications of Section 6, however, we shall need the entire series of $D^{(k)}$ implemented by *OUT* vectors.

5. Reducing the Space. Both the preparation of D_x and its subsequent refinements need $\Theta(n^2)$ space. Procedure *Skeleton-Tree* needs $\Theta(n^2)$ space due to the array *BB*, which is used at each iteration q to partition the composite labels (*TID*s) into equivalence classes. In any refining stage, the nest of each node v needs a distinct array of n locations for partitioning the split-labels of the nodes in the nest into equivalence classes. In this section we show that both problems can be solved using only $\Theta(n^{1+\varepsilon})$ space, for any $0 < \varepsilon \leq 1$, at the expense of a corresponding slow-down proportional to $1/\varepsilon$.

We analyze the procedure *Skeleton-Tree* first. Consider some substring w of x of length 2^q, with $q > 0$, and let $w = w_1 w_2$ with $|w_1| = |w_2| = 2^{q-1}$. Let N_1 and N_2 be the *ID*s assigned by the procedure to w_1 and w_2, respectively. Recall that each

of N_1, N_2 is an integer between 1 and n. The difficulty in creating the *ID* for w is that the pair (N_1, N_2) may assume n^2 values.

We show how to solve this problem using only $\Theta(n^{1+\varepsilon})$ space. We assume for simplicity that n^ε is an integer, but it is easy to generalize our solution to the cases where n^ε is not an integer. We focus on computing the *ID* of the string w above. The same manipulations are performed in parallel for all substrings of x of length 2^q. The idea is to express N_2 by its representation in the base n^ε. The coefficients $(a_1, a_2, \ldots, a_{1/\varepsilon})$ (least-significant coefficient first) of this representation are easily computed in $1/\varepsilon$ steps as follows:

for $i = 1$ **to** $1/\varepsilon$ **do**
begin
$a_i \leftarrow N_2 \bmod n^\varepsilon$
$N_2 \leftarrow \left\lfloor \frac{N_2}{n^\varepsilon} \right\rfloor$
end

Iteration q of *Skeleton-Tree* is now modified to contain $1/\varepsilon$ subiterations. The input to subiteration δ, $\delta = 1, \ldots, 1/\varepsilon$, is as follows:

(i) An *ID* for the pair consisting of the left substring and the $\delta - 1$-tuple $(a_1, \ldots, a_{\delta-1})$. This *ID* is a number between 1 and n.
(ii) The *ID* a_δ, i.e., a number between 0 and $n^\varepsilon - 1$.

The output of subiteration δ is an *ID* for the pair consisting of the left substring and the δ-tuple $(a_1, \ldots, a_\delta)$. This *ID* is a number between 1 and n.

The concurrent-write contests that take place within any subiteration of iteration q of the *Skeleton-Tree* procedure are similar to the original ones. The only difference is that now an auxiliary array of size $(n+1) \times n^\varepsilon$ suffices. Details are left to the reader. For any fixed $0 < \varepsilon \leq 1$ the total space requirements are bounded by $O(n^{1+\varepsilon})$ and the running time by $O((1/\varepsilon) \log n) = O(\log n)$.

Our space reduction technique extends easily to the refining stages. We outline the main changes and omit the tedious details. With reference to the generic intermediate tree $D^{(k)}$, we focus on the processors that handle the nest of some node v. Recall that, in order to refine this nest, the processors partition their underlying nodes into equivalence classes, according to the first component of the split-labels. For this purpose, a row of *BB* was used in our original construction, namely, the row assigned to the representative processor of the nest. Assume instead that processor p_i, $i = 1, \ldots, n$, is assigned only an array *LITTLE-BB*$_i$ consisting of n^ε locations of the common memory, and let p_j be the representative of the nest of v. We perform the partition of the nest in $1/\varepsilon$ subiterations as follows. First, all processors in the nest compute the representation of the first component of their split-labels in the base n^ε. There are n^ε possible values for the first coefficient of this representation. Thus, the processors in the nest can partition themselves in n^ε classes through a concurrent-write contest on *LITTLE-BB*$_j$. In this way, each class elects a representative processor. The *LITTLE-BB* arrays associated with these representatives is similarly used to obtain a second

refinement of the classes. This refinement is based on the second coefficients in the representations of the split-labels in base n^{ε}. It should be clear how to proceed with the remaining $1/\varepsilon - 2$ subiterations. For any fixed $0 < \varepsilon \leq 1$ the total space requirements are bounded by $O(n^{1+\varepsilon})$ and the running time by $O((1/\varepsilon)\log n) = O(\log n)$.

If the suffix tree is implemented by OUT_v vectors, as needed in the next section, it would require $\Theta(n^2)$ space. However, we can reduce the space to $\Theta((1/\varepsilon)n^{1+\varepsilon}) = \Theta(n^{1+\varepsilon})$ ising the ideas of the space reduction described above.

6. Applications. In this section we describe some applications of our parallel suffix tree construction in the design of efficient parallel algorithms.

PROBLEM 1. *On-line string matching.* Suppose a string $x = x_1, \ldots, x_{n-1}, \#$ is given in advance (for preprocessing). Answer as fast as possible (on-line) queries of the form: "Does the string $z = z_1, \ldots, z_m$ (the *pattern*) occur in x?"

SOLUTION.

Preprocessing. Construct the suffix tree of $x\#$. In the course of the computation we save:

(1) The $\log n$ *BB*s used in $\log n$ iterations of the procedure *Skeleton-Tree.*
(2) All the intermediate trees $D^{(k)}$, $k = \log n - 1, \ldots, 0$. Each of these intermediate trees is implemented by the vectors OUT_v, defined in Section 4.

The computation of this step takes $O(\log n)$ time using n processors.

On-line Processing of the Queries.

Step 1. Recall that in Section 2 we computed $ID_i[q]$ $(i = 1, \ldots, n;\ q = 0, \ldots, \log n)$ for the string $x\#$. The value $ID_i[q]$ is a unique name of the substring $x_i, \ldots, x_{i+2^q-1}$, where $ID_i[q] = ID_j[q]$ if $x_i, \ldots, x_{i+2^q-1} = x_j, \ldots, x_{j+2^q-1}$. We start the on-line processing by naming some of the substrings in the pattern z. For $q = 0, \ldots, \lfloor \log m \rfloor$, the substrings we are naming are all substrings whose length is 2^q which start at positions i, where i is a multiple of 2^q and $i + 2^q \leq m$. The names are stored in the vectors $PID_i[q]$ (i.e., $PID_i[q]$ is the unique name of the substring $z_i, \ldots, z_{i+2^q-1}$). The naming is done such that if two substrings of length 2^q, one in z and the other in $x\#$, are equal then their names are equal too. For this, we compute the *PID* labels using the same *BB*s used in the *Skeleton-Tree* procedure. (These *BB*s are saved in the preprocessing stage.)

Step 2. Let $PID_1[\lfloor \log m \rfloor]$ (that is, the name of the prefix of z whose length is $2^{\lfloor \log m \rfloor}$) be k. Observe that if none of the $ID_i[\lfloor \log m \rfloor]$ is equal to k then the prefix of z whose length is $2^{\lfloor \log m \rfloor}$ does not occur in x. We conclude that the answer to the query is NO (i.e., z does not occur in x). Suppose $k = ID_i[\lfloor \log m \rfloor]$ for some $1 \leq i \leq n-1$. We check whether $NODE_k[\lfloor \log m \rfloor]$ appears in $D^{(\log n - 1)}$. Note that $NODE_k[\lfloor \log m \rfloor]$ will not appear in $D^{(\log n - 1)}$ if and only if all the

substrings of x whose prefix of length $2^{\lfloor \log m \rfloor}$ is the same as the prefix of z also have the same prefix of length $2^{\lfloor \log m \rfloor + 1}$. If $NODE_k[\lfloor \log m \rfloor]$ appears in $D^{(\log n - 1)}$ then we are guaranteed that it will also appear in $D^{(\lfloor \log m \rfloor)}$ and we proceed to step 3. This is because all the refinements $D^{(\log n - 1)}, \ldots, D^{(\lfloor \log m \rfloor)}$ deal only with substrings whose length is greater than $2^{\lfloor \log m \rfloor}$. Otherwise, i.e., $NODE_k[\lfloor \log m \rfloor]$ does not appear in $D^{(\log n - 1)}$, we check whether z is equal to $x_k, \ldots, x_{k+m-1}$ letter by letter. This can be done in $\log m$ time using $m/\log m$ processors. The answer to the query is YES if and only if the two strings are equal.

Step 3. We find a node v in the suffix tree such that z is a prefix of $W(v)$ (if such node exists). For this, we use the vectors $PID_i[q]$ of step 1 and the $D^{(q)}$ trees, $q = \lfloor \log m \rfloor - 1, \ldots, 0$ of the preprocessing. Node v is found using some notion of binary search in $\lfloor \log m \rfloor$ iterations.

Iteration q ($q = \lfloor \log m \rfloor - 1, \ldots, 0$). Let v and z' be the input parameters of iteration q. (For iteration $\lfloor \log m \rfloor - 1$, $v = NODE_k[\lfloor \log m \rfloor]$ and z' is the suffix of z starting at position $2^{\lfloor \log m \rfloor} + 1$.) The invariant property satisfied in all the iterations is that v is a node in $D^{(q+1)}$ and z' is a substring whose length is less than 2^{q+1}. Our goal is to check whether z' follows an occurrence of $W(v)$. We work on $D^{(q)}$. There are two possibilities:

1. The node v appears in $D^{(q)}$. Possibility 1 has two subpossibilities:
   1.1. 2^q is larger than the length of z'. In this case we do nothing and the input parameters of the present iteration become the input parameters of the next iteration.
   1.2. 2^q is less than or equal to the length of z'. Assume that z' starts at position j of z and b is the value stored in $PID_j[q]$. If the entry $OUT_v[b]$ is empty then z does not occur in x. Otherwise, the input parameters of the next iteration will be the suffix of z' starting at position $2^q + 1$ and the node pointed to by $OUT_v[b]$.
2. The node v does not appear in $D^{(q)}$. This means that v had only one son in $\bar{D}^{(q+1)}$ and so it was omitted from $D^{(q)}$ (in step 2 of refining $\bar{D}^{(q+1)}$). Let μ be the single son of v in $\bar{D}^{(q+1)}$. Possibility 2 has two subpossibilities:
   2.1. 2^q is larger than the length of z'. Assume that the $LABEL$ of μ in $D^{(q)}$ is (i, l). In this case z' occurs in x if and only if z' is a prefix of $x_{i+l-2^q+1}, \ldots, x_{i+l}$. We check this letter by letter in $\log m$ time using $m/\log m$ processors.
   2.2. 2^q is less or equal to the length of z'. We compare $ID_{i+l-2^q+1}[q]$ (the unique name of $x_{i+l-2^q+1}, \ldots, x_{i+l}$) to the unique name of the prefix of z' whose length is 2^q. If these names are different then z does not occur in x. Otherwise, the input parameters of the next iteration will be the suffix of z' starting at position $2^q + 1$ and the node μ.

Remarks. (a) We did not initialize the vectors OUT_v, therefore it could be that we will get a wrong positive answer. To avoid mistakes, every time we get a positive answer we explicitly check whether z really appears in x at the position

given in the answer. This can be done in $\lfloor \log m \rfloor$ time using $m/\log m$ processors as a last step.

(b) The on-line computation can be extended to obtain additional information about z. For example:

(1) What is the number of occurrences of z in x?
(2) In case there is more than one occurrence, what is the starting position of the first (or last or all) occurrence(s) of z in x?
(3) What is the longest prefix of z which occurs in x?

Complexity. The preprocessing takes $O(\log n)$ time using n processors. Answering a query takes $O(\log m)$ time using $m/\log m$ processors.

PROBLEM 2. *Finding the longest repeated substring in a string.* Given a string x find the longest substring which occurs in x more than once.

SOLUTION. $W(v)$ is defined in Section 2. Let $|W(v)|$ be the length of $W(v)$.

Step 1. Construct the suffix tree of $x\#$ and find $|W(v)|$ of each node v.

Step 2. Find the internal node v with the maximum $|W(v)|$ field. The substring represented by the path from the root to v is the longest repeated substring in x.

Step 2 can be carried out using the parallel algorithm for finding the maximum given in [SV-81].

Complexity. Step 1 takes $O(\log n)$ time using n processors. Step 2 takes $O(\log \log n)$ time using $n/\log \log n$ processors.

PROBLEM 3. *Approximate string matching.* Suppose a string x, a pattern z, and a parameter k are given. (Let n (resp. m) be the length of x (resp. z).) Find occurrences of z in x with at most k differences. We distinguish three types of differences:

(a) A letter in z corresponds to a different letter in x.
(b) A letter in z corresponds to "no letter" in x.
(c) A letter in x corresponds to "no letter" in z.

SOLUTION. [LV-86] gave both a serial and a parallel algorithm for the problem. This paper enables us to design an alternative parallel algorithm which essentially consists of parallelizing the serial algorithm of [LV-86]. The alternative parallel algorithm is based on both the parallel prefix tree construction, of this paper, and parallel algorithm for answering Lowest Common Ancestor (LCA) queries of [ScV-87]. In order to keep this presentation short we refrain from describing this alternative algorithm in detail. This alternative parallel algorithm for the approximate string matching problem runs in time $O(k+\log n)$ using $n+m$ processors. Note that the parallel algorithm of [LV-86] consists of two parts: (1) analysis of the pattern and (2) analysis of the text. Part 1 runs in $O(\log m)$ time

using m^2 processors. Part 2 runs in $O(k+\log m)$ time using n processors. So, comparison of the performance of these two parallel algorithms depends on the relative values of n and m and also on whether the pattern is given in advance for preprocessing.

Acknowledgment. We are grateful to Zvi Galil for stimulating discussions and helpful comments, to C. L. Liu and J. S. Vitter for their insightful editorial assistance, and to the referees for their careful reviews.

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.

[Ap-84] A. Apostolico, On context-constrained squares and repetitions in a string, *RAIRO Theoretical Informatics*, **18** (1984), 147–159.

[Ap-85] A. Apostolico, The myriad virtues of subword trees, in A. Apostolico and Z. Galil (editors), *Combinatorial Algorithms on Words*, NATO ASI Series, Series F: Computer and System Sciences, Vol. 12, Springer-Verlag, Berlin, 1985, pp. 85–96.

[AG-86] A. Apostolico and R. Giancarlo, The Boyer-Moore-Galil string searching strategies revisited, *SIAM Journal on Computing*, **15** (1986), 98–105.

[AI-86] A. Apostolico and C. Iliopoulos, Parallel log-time construction of suffix trees, CSD TR 632, Department of Computer Science, Purdue University, Sept. 1986.

[AP-83] A. Apostolico and F. P. Preparata, Optimal off-line detection of repetitions in a string, *Theoretical Computer Science*, **22** (1983), 297–315.

[AP-85a] A. Apostolico and F. P. Preparata, Structural properties of the string statistics problem, *Journal of Computer and System Sciences*, **31** (1985), 394–411.

[AP-85b] A. Apostolico and F. P. Preparata, Data structures and algorithms for the string statistics problem, CSD TR 541, Department of Computer Science, Purdue University, Sept. 1985.

[BH-82] A. Borodin and J. E. Hopcroft, Routing, merging and sorting on parallel models of computation, *Journal of Computer and System Science*, **30** (1985), 130–145.

[CV-86a] R. Cole and U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, *Information and Control*, **70** (1986), 32–53.

[CV-86b] R. Cole and U. Vishkin, Approximate and exact parallel scheduling with applications to list, tree, and graph problems, *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, 1986, pp. 478–491.

[FL-80] M. Fisher and L. Ladner, Parallel prefix computation, *Journal of the Association for Computing Machinery*, **27** (1980), 831–838.

[Ga-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, Berlin, 1985, pp. 1–10.

[KMR-72] R. M. Karp, R. E. Miller, and A. L. Rosenberg, Rapid identification of repeated patterns in strings, trees, and arrays, *Proceedings of the 4th ACM Symposium on Theory of Computing*, 1972, pp. 125–136.

[Kr-83] C. P. Kruskal, Searching, merging, and sorting in parallel computation, *IEEE Transactions on Computers*, **32** (1983), 942–946.

[LSV-86] G. M. Landau, B. Schieber, and U. Vishkin, Parallel construction of a suffix tree, TR-53/86, Department of Computer Science, Tel Aviv University, 1986, and also *Proceedings of the 14th ICALP*, Lecture Notes in Computer Science, Vol. 267, Springer-Verlag, Berlin, 1987, pp. 314–325.

[LV-86] G. M. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching, *Proceedings of the 18th ACM Symposium on Theory of Computing*, 1986, pp. 220–230.

[Mc-76] E. M. McCreight, A space-economical suffix tree construction algorithm, *Journal of the Association for Computing Machinery*, **23** (1976), 262–272.

[ScV-87] B. Schieber and U. Vishkin, Parallel computation of lowest common ancestor in trees, TR-63/87, Department of Computer Science, Tel Aviv University, 1987.

[SV-81] Y. Shiloach and U. Vishkin, Finding the maximum, merging, and sorting in a parallel model of computation, *Journal of Algorithms*, **2** (1981), 88–102.

[Va-75] L. G. Valiant, Parallelism in comparison problems, *SIAM Journal on Computing*, **4** (1975), 348–355.

[Vi-83] U. Vishkin, Synchronous parallel computation—a survey, TR-71, Department of Computer Science, Courant Institute, New York University, 1983.

[Vi-84] U. Vishkin, Randomized speed-ups in parallel computation, *Proceedings of the 16th ACM Symposium on Theory of Computing*, 1984, pp. 230–239.

[We-73] P. Weiner, Linear pattern matching algorithm, *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, 1973, pp. 1–11.