1992

On-Line Serial and Parallel String Searching

Alberto Apostolico

Report Number:
92-090
ON-LINE SERIAL AND PARALLEL STRING SEARCHING

Alberto Apostolico

CSD-TR-92-090
November 1992
On-line Serial and Parallel String Searching

Alberto Apostolico
Department of Computer Science
Purdue University
West Lafayette, IN 47907, USA

Purdue CS-TR 92-090
November 1992

Abstract

String searching is a pervasive primitive of computation. In on-line string searching, the text string is given in advance, but it is required that, as soon as one finishes reading a pattern, it must be possible to decide whether or not it occurs in the text. This paper surveys serial and parallel methods for on-line string searching. The auxiliary structures used are also suited to a host of other applications.

Key Words: Combinatorial Algorithms on Words, String Matching, Suffix Trees, Universal (Sub)string Searching, Parallel Computation, CRCW PRAM, Lexicographic Order, Squares and Repetitions in a String.

AMS subject classification: 68C25

*Work supported in part by the National Research Council of Italy, by NSF Grants CCR-89-00305 and CCR-92-01078, by NIH Library of Medicine Grant RO1 LM05118, by AFOSR Grant 90-0107, and by NATO Grant CRG 900293.
1 Introduction

There are various serial and parallel methods to perform exact string searching in a number of operations proportional to the total length of the input. Even though such a performance is optimal, such algorithms do not exhaust the treatment of exact searches: in many applications, searches for different, a-priori unknown patterns are performed on a same text or group of texts. It seems natural to ask whether these cases can be handled better than by plain reiteration of the procedures studied so far. As an analogy, consider the classical problem of searching for a given item in a table with \( n \) entries. In general, \( n \) comparisons are both necessary and sufficient for this task. If we wanted to perform \( k \) such searches, however, it is no longer clear that we need \( kn \) comparisons. Our table can be sorted once and for all at a cost of \( O(n \log n) \) comparisons, after which binary search can be used. For sufficiently large \( k \), this approach outperforms that of the \( k \) independent searches.

In this review, we shall see that the philosophy subtending binary search can be fruitfully applied to string searching. Specifically, the text can be pre-processed once and for all in such a way that any query concerning whether or not a pattern occurs in the text can be answered in time proportional to the length of the pattern. It will also be possible to locate all the occurrences of the pattern in the text at an additional cost proportional to the total number of such occurrences. We call this type of search \textit{on-line}, to refer to the fact that as soon as we finish reading the pattern we can decide whether or not it occurs in our text. As it turns out, the auxiliary structures used to achieve this goal are well suited to a host of other applications.

2 Subword trees

There are several, essentially equivalent digital structures supporting efficient on-line string searching. Here, we base our discussion on a variant known as suffix tree. It is instructive to discuss first a simplified version of suffix trees, which we call \textit{expanded suffix tree}. This version is not the most efficient from the standpoint of complexity, but it serves a few pedagogical purposes, among which that of clearly exposing the relationship between subword trees and finite automata.

Let \( x \) be a string of \( n - 1 \) symbols over some alphabet \( \Sigma \) and \( \$ \) an extra character not in \( \Sigma \). The \textit{expanded suffix tree} \( T_x \) associated with \( x \) is a digital search tree collecting all suffixes of \( x\$ \). Specifically, \( T_x \) is defined as follows.

1. \( T_x \) has \( n \) leaves, labeled from 1 to \( n \).

2. Each arc is labeled with a symbol of \( \Sigma \cup \{\$\} \). For any \( i, 1 \leq i \leq n \), the concatenation of the labels on the path from the root of \( T_x \) to leaf \( i \) is precisely the suffix \( \text{suffix}_i = x_1x_2x_{i+1}...x_{n-1}\$. 


3. For any two suffixes $suf_i$ and $suf_j$ of $x\$, if $w_{ij}$ is the longest common prefix that $suf_i$ and $suf_j$ have in common, then the path in $T_x$ relative to $w_{ij}$ is the same for $suf_i$ and $suf_j$.

An example of expanded suffix tree is given in Figure 1.
The tree can be interpreted as the state transition diagram of a deterministic finite automaton where all nodes and leaves are final states, the root is the initial state, and the labeled arcs, which are assumed to point downwards, represent part of the state-transition function. The state transitions not specified in the diagram lead to a unique non-final sink state. Our automaton recognizes the (finite) language consisting of all substrings of string \( x \). This observation clarifies also how the tree can be used in an on-line search: letting \( y \) be the pattern, we follow the downward path in the tree in response to consecutive symbols of \( y \), one symbol at a time. Clearly, \( y \) occurs in \( x \) if and only if this process takes to a final state. In terms of \( T_x \), we say that the locus of a string \( y \) is the node \( \alpha \), if it exists, such that the path from the root of \( T_x \) to \( \alpha \) is labeled \( y \).

**Fact 1** A string \( y \) occurs in \( x \) if and only if \( y \) has a locus in \( T_x \).

The implementation of Fact 1 takes \( O(t \cdot |y|) \) character comparisons, where \( t \) is the time necessary to traverse a node, which is constant for a finite alphabet. Note that this only answers whether or not \( y \) occurs in \( x \).

**Fact 2** If \( y \) has a locus \( \alpha \) in \( T_x \), then the occurrences of \( y \) in \( x \) are all and only the labels of the leaves in the subtree of \( T_x \) rooted at \( \alpha \).

Thus, if we wanted to know where \( y \) occurs, it would suffice to visit the subtree of \( T_x \) rooted at node \( \alpha \), where \( \alpha \) is the node such that the path from the root of \( T_x \) to \( \alpha \) is labeled \( y \). Such a visit requires time proportional to the number of nodes encountered, and the latter can be \( O(n^2) \) on the expanded suffix tree. This is as bad as running an offline search naively, but we will see shortly that a much better bound is possible.

An algorithm for the construction of the expanded \( T_x \) is readily organized (see Figure 2). We start with an empty tree and add to it the suffixes of \( x\$ \) one at a time. Conceptually, the insertion of suffix \( suf_i \) (\( i = 1, 2, ..., n \)) consists of two phases. In the first phase, we search for \( suf_i \) in \( T_{i-1} \). Note that the presence of \( \$ \) guarantees that every suffix will end in a distinct leaf. Therefore, this search will end with failure sooner or later. At that point, though, we will have identified the longest prefix of \( suf_i \) that has a locus in \( T_{i-1} \). Let \( head_i \) be this prefix and \( \alpha \) the locus of \( head_i \). We can write \( suf_i = head_i \cdot tail_i \) with \( tail_i \) nonempty. In the second phase, we need to add to \( T_{i-1} \) a path leaving node \( \alpha \) and labeled \( tail_i \). This achieves the transformation of \( T_{i-1} \) into \( T_i \).

We will assume that the first phase of \( \text{INSERT} \) is performed by a procedure \( \text{FINDHEAD} \), which takes \( suf_i \) as input and returns a pointer to the node \( \alpha \). The second phase is performed by a procedure \( \text{ADDPATH} \), which receives such a pointer and directs a path from node \( \alpha \) to leaf \( i \). The details of these procedures are left for an exercise.

**Theorem 1** The procedure \( \text{BUILDTREE} \) takes time \( \Theta(n^2) \) and linear space.
procedure BUILDTREE( x, Tx )
begin
    T0 ← Ø;
    for i = 1 to n do Ti ← INSERT(sufi, Ti-1);
    Tx = Tn;
end

Figure 2: Building an expanded suffix tree

Proof: The procedure performs n calls to INSERT. The ith such call requires time proportional to the length n - i + 1 of sufi. Hence the total charge is proportional to \( \sum_{i=1}^{n} (n + 1 - i) = \sum_{i=1}^{n} i = n(n + 1)/2 \). □

It is instructive to examine the cost of BUILDTREE in terms of the two constituent procedures of INSERT. If the symbols of x are all different, then Tx contains \( \Theta(n^2) \) arcs. The procedure FINDHEAD only charges linear time overall, and the heaviest charges come from ADDPATH. At the other extreme, consider x = a^n-1. In this case, ADDPATH charges linear time overall and the real work is done by FINDHEAD.

It is easy to reduce the work charged by ADDPATH by resorting to a more compact representation of Tx. Specifically, we can collapse every chain formed by nodes with only one child into a single arc, and label that arc with a substring, rather than with a symbol of x$. Such a compact version of Tx has at most n internal nodes, since there are n + 1 leaves in total and every internal node is branching. The compact version of the tree of Figure 1 is in Figure 3. Clearly, the two versions are equivalent for our purposes, and it takes little to adapt the details of BUILDTREE in order to fit the new format.

With the new convention, the tree for a string formed by all different symbols only requires 1 internal node, namely, the root. Except for arc-labeling, the construction of such a tree is performed in linear time, since ADDPATH spends now constant time per suffix. However, there is no improvement in the management of the case x = a^n-1, in which FINDHEAD still spends \( \Theta(n^2) \) time.

While the topology of the tree requires now only \( O(n) \) nodes and arcs, each arc is labeled with a substring of x$. We have seen that the lengths of these labels may be \( \Theta(n^2) \) (think again of the tree for a string formed by all different symbols). Thus, as long as this labeling policy is maintained, Tx will require \( \Theta(n^2) \) space in the worst case, and it is clearly impossible to build a structure requiring quadratic space in less that quadratic worst-case time. Fortunately, a more efficient labeling is possible which allows us to store Tx in linear space. For this, it is sufficient to encode each arc label into a suitable pair of pointers in the form \([i, j]\) to a single common copy of x. For instance, pointer i denotes the starting position of the label and j the end. Now Tx takes linear space and it makes
sense to investigate its construction in better than quadratic time.

As already seen, the time consuming operation of INSERT is in the auxiliary procedure FINDHEAD. For every $i$, this procedure starts at the the root of $T_{i-1}$ and essentially locates the longest prefix $head_i$ of $suf_i$ that is also a prefix of $suf_j$ for some $j < i$. Note that $head_i$ will no longer necessarily end at a node of $T_{i-1}$. When it does, we say that $head_i$ has a proper locus in $T_{i-1}$. If $head_i$ ends inside an arc leading from some node $\alpha$ to some node $\beta$, we call $\alpha$ the contracted locus and $\beta$ the extended locus of $head_i$. We use the word locus to refer to the proper or extended locus, according to the case. It is trivial to upgrade FINDHEAD in such a way that the procedure creates the proper locus of $head_i$ whenever such a locus does not already exist. Note that this part of the procedure only requires constant time.
3 McCreight's algorithm

The discussion of the previous section embodies the obvious principle that the construction of a digital search tree for an arbitrary set of words \( \{w_1, w_2, ..., w_k\} \) cannot be done in time better than the \( \sum_{i=1}^{k} |w_i| \) in the worst case. This seems to rule out a better-than-quadratic construction for \( T_z \), even when the tree itself is in compact form. However, the words stored in \( T_z \) are not unrelated, since they are all suffixes of a same string. This simple fact has the following important consequences.

**Lemma 1** For any \( i, 1 \leq i \leq n, |head_{i+1}| \geq |head_i| - 1 \)

**Proof:** Assume the contrary, i.e., \( |head_{i+1}| < |head_i| - 1 \). Then, \( head_{i+1} \) is a substring of \( head_i \). By definition, \( head_i \) is the longest prefix of \( suf_i \) that has another occurrence at some position \( j < i \). Let \( x_jx_{j+1}...x_{j+|head_i|-1} \) be such an occurrence. Clearly, any substring of \( head_i \) has an occurrence in \( x_jx_{j+1}...x_{j+|head_i|-1} \). In particular, \( x_{j+1}x_{j+2}...x_{j+|head_i|-1} = x_{i+1}x_{i+2}...x_{i+|head_i|-1}, \) hence \( x_{i+1}x_{i+2}...x_{i+|head_i|-1} \) must be a prefix of \( head_{i+1} \).

**Lemma 2** Let \( w = ay, \) with \( a \in \Sigma \) and \( y \in \Sigma^* \). If \( w \) has a proper locus in \( T_z \), so does \( y \).

**Proof:** Since every node of \( T_z \) is branching, then the fact that \( w \) has a proper locus in \( T_z \) means that there are at least substrings of \( x \) in the form \( wa \) and \( wb \) with \( a, b \in \Sigma \) and \( a \neq b \). But then \( ya \) and \( yb \) are also substrings of \( x \).

Note that the converse of Lemma 2 is not true. Lemmas 1 and 2 are very helpful. Assume we have just inserted \( suf_i \) into \( T_{i-1} \). Because \( suf_i = head_i \cdot tail_i \) with \( tail_i \) nonempty, we are guaranteed that \( head_i \) has a proper locus, say \( \alpha \) in \( T_i \). Let \( head_i = ay \). Clearly, there is a path labeled \( y \) in \( T_i \). Assume that we can reach instantaneously the end of this path. This might be at a node or in the middle of an arc. Lemma 1 tells us that \( head_{i+1} \) is not shorter than this path. Once we are at that point we only need go further into the tree in response to the symbols of \( suf_{i+1} \) that follow \( y_i \), until we fall off the tree again. Let \( head_{i+1} = y_i z_{i+1} \), where \( z_{i+1} \) is possibly empty. Clearly, we only need to do work proportional to \( z_{i+1} \). Having found \( head_{i+1} \), we can invoke the same principle and write \( head_{i+2} = a'y_{i+1} \) so that \( head_{i+2} = y_{i+1} z_{i+2} \), and so on.

**Lemma 3** \( \sum_{i=1}^{n+1} |z_i| = n \)

**Proof:** The \( z_i \)'s are non-overlapping.

Lemma 3 suggests that FINDHEAD be regarded as consisting of two stages. With reference to the insertion of \( suf_{i+1} \), Stage 1 consists of finding the end of the path to \( y_i \), and Stage 2 consists of identifying \( z_{i+1} \). For reasons that will become apparent in the sequel, we refer to Stage 2 as the scanning. Lemma 3 shows that all executions of
scanning take amortized linear time. Thus the main task is to perform Stage 1 with the
same amortized efficiency as Stage 2.

Let us add to the structure of $T_x$ some new links called suffix links and defined as
follows: For every string $w = ay$ having a proper locus in $T_x$, there is a link directed from
the locus of $w$ to the locus of $y$. It would be nice to have that, at the inception of every
iteration of BUILDTREE, every node of the tree produced thus far has a defined suffix
link. In fact, assume that, upon completion of the insertion of $su_i$, both $head_i$ and $y_i$
had proper loci in $T_i$. Following the suffix link assigned to the locus $\alpha$ of $head_i$ we would
reach instantaneously the locus of $y_i$. In other words, Stage 1 would require constant time
per iteration.

Unfortunately, there are two difficulties. The first one is that lemma 2 tells us that $y_i$
has a proper locus in $T_x$, but it says nothing of $T_i$. In other words, $y_i$ is not guaranteed
to have a proper locus in $T_i$. The second difficulty is that, upon completion of $T_i$, even if
$y_i$ had a proper locus in $T_i$, it might be impossible to reach it immediately from the locus
$\alpha$ of $head_i$, for the simple reason that $\alpha$ was just created as part of the $i$th iteration of
BUILDTREE. In conclusion, we cannot maintain as an invariant that every node of $T_i$
has a defined suffix link. However, we can maintain the next best thing, namely:

**Invariant 1** In $T_i$, every node except possibly the locus of $head_i$ has a defined suffix link.

At the beginning, the suffix link of the root points to the root itself. Assume that we
have just found the locus of $head_i$. By Invariant 1, $Father[\alpha]$ has a defined suffix link. Let
$head_i = ay_i = aw_\alpha s_\alpha$ where $aw_\alpha$ is the (possibly empty) prefix of $y_i$ having $Father[\alpha]$ as its
proper locus. By following the suffix link from $Father[\alpha]$, we thus reach the proper locus
$\gamma$ of $w_\alpha$. Once at node $\gamma$, we know that we need to go down in the tree for at least $|s_\alpha|$ symbols, by virtue of Lemma 1. This phase is called rescanning, since we have already
seen the symbols of $s_\alpha$. Before examining the mechanics of rescanning, we point out that
it may end up in one of two possible ways:

1. $y_i = w_is_i$ has a proper locus in $T_i$.

2. $y_i = w_is_i$ has an extended locus in $T_i$.

Case 1 is relatively easy. All we have to do is to set the suffix link from $\alpha$ to the locus
$\gamma$ of $y_i$, and initiate the scanning from this node. Case 2 is more elaborate. Note that
Invariant 1 prescribes that at the end of this pass there be a suffix link defined from $\alpha$ to
the proper locus of $y_i$. Since such a locus did not exist, we have to introduce it at this
moment. But we are in the middle of an arc, and splitting an arc with a node having
only one child might infringe our convention on the structure of $T_x$! The following lemma
ensures that no such infringement will take place.
Lemma 4 If \( y_i \) does not have a proper locus in \( T_i \), then head_{i+1} = y_i.

**Proof:** Exercise. \( \square \)

In principle, we may design the rescanning along the same lines as the scanning. Unlike the \( z_i \) substrings involved in scanning, however, the \( s_i \) substrings involved in rescanning present mutual overlaps. This is undesirable, since it plays havoc with the linear time complexity. A closer look reveals a significant difference between scanning and rescanning: in rescanning, we know beforehand the length of the substring \( s \) being rescanned. This allows us to rescan in time proportional to the number of nodes traversed in rescanning, rather than to the length \( |s| \) itself. At first sight, it is not clear how this would induce a savings, since the number of such nodes can be \( \Theta(|s|) \). However, we will show that the total number of nodes involved in rescanning is linear in \( |x| \). Before getting to that point, let us refine the details of rescanning.

Suppose we reached node \( \gamma \), the locus of \( w_i \), and let \( \bar{a} \) be the first symbol of \( s_i \). There is precisely one arc leaving \( \gamma \) with a label that starts with \( \bar{a} \). Let \( \gamma_1 \) be the child of \( \gamma \) along this arc. By comparing \( |s_i| \) and the length of the label of the arc \((\gamma, \gamma_1)\) we can decide in constant time whether the locus of \( y_i = w_i; s_i \) is in the middle of this arc, precisely on \( \gamma_1 \) or further below. In the first two cases the rescanning is finished, in the third case, we move to \( \gamma_1 \) having rescanned a prefix of \( s_i \), and need still to rescan a suffix \( s_i' \) of \( s_i \). We proceed in the same way from \( \gamma_1 \), thus finding a descendant \( \gamma_2 \) of \( \gamma_1 \), and so on. The time spent at each of the nodes \( \gamma, \gamma_1, \gamma_2, \ldots \) is constant, whence rescanning takes time linear in the number of nodes traversed.

Lemma 5 The number of intermediate nodes encountered in rescanning thru all iterations of BUILDTREE is \( O(n) \).

**Proof:** Let \( res_i \) be defined as the shortest suffix of \( xS \) to which the rescanning and scan operations are confined during the \( i \)th iteration of BUILDTREE. Observe that for every intermediate node \( \gamma_f \) encountered during the rescan of \( s_i \), there will be a nonempty string which is contained in \( res_i \) but not in \( res_{i+1} \). Therefore, \( |res_{i+1}| \) is at most \( |res_i| - int_i \), where \( int_i \) is the number of intermediate nodes encountered while rescanning at iteration \( i \). By repeated substitutions we see that \( \sum_{i=1}^{n+1} int_i \) is at most \( n \), since \( |res_{n+1}| = 0 \) and \( |res_0| = 0 \). Thus, the number of nodes encountered during the rescanning is at most \( n \). \( \square \)

In conclusion, we can formulate the following

**Theorem 2** The suffix tree in compact form for a string of \( n \) symbols can be built in \( O(t \cdot n) \) time and \( O(n) \) space, where \( t \) is the time needed to traverse a node.
4 Storing suffix trees

When the alphabet $\Sigma$ is a constant independent of $n$, the factor $t$ in Theorem 2 is also a constant. It is desirable to detail how the constructions of the previous sections handle the cases where $|\Sigma|$ is not a constant. For this purpose, we must address the issue of the memory allocations of suffix trees. This is done in this section.

In some applications, $T_x$ needs only be traversed bottom-up. This occurs, for instance, in connection with computations of the squares in a string, or in computing substring statistics, etc. In all these cases, a satisfactory representation of the tree is achieved by letting each node have precisely one pointer, directed to its father. This node format does not pose any problem in allocation irrespective of the size of $\Sigma$.

For problems like on-line searches, which we used as motivation in our discussion, we need to traverse the tree downwards from the root, and thus we need that edges be directed from each node to its children. The number of edges leaving a node is bounded above by $|\Sigma|$, and $|\Sigma|$ can be $\Theta(n)$. In other words, even though there are $O(n)$ arcs in $T_x$ irrespective of the size of $\Sigma$, the number of arcs leaving a specific node can assume any value from 2 to $\Theta(n)$. This poses a problem of efficiently formatting the nodes of $T_x$. Before addressing this point, we recall that, in addition to the edges leaving it, each node of $T_x$ must also store appropriate branching labels for all the downward edges originating from it. Such labels are needed during the construction of $T_x$, and they also drive, e.g., the downward search in $T_x$ of any string $w$. Earlier in this Chapter, we stipulated that each edge be labeled with a pair of integers pointing to a substring of $x$. In order to leave a node towards one of its children, however, we need to know the first character of such a substring. To fix the ideas, let $(i, j)$ be the label of an edge $(a, b)$. We may use our knowledge of $i$ to access the character $x_i$. Alternatively, we could add to the pair $(i, j)$ the symbol of $\Sigma$ that corresponds to $x_i$. The two approaches are different, since we need $\log |\Sigma|$ bits to identify a symbol and $\log n$ bits to identify a position of $x$.

The set of branching labels leaving each internal node of $T_x$ can be stored using a linear list, a binary trie, or an array. Resorting to arrays supports, say, searching for a word $w$ in $T_x$ in time $O(|w|)$, but requires space $\Theta(|\Sigma|n)$ or $\Theta(n^2)$, depending on the labeling convention adopted, to store $T_x$. Note that the initialization of the overall space allocated seems to require quadratic time. Fortunately, techniques are available to initialize only the space which is actually used. We leave this as an exercise. 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 $O(|w|\log |\Sigma|)$ and $O(|w|\log n)$, respectively. Such bounds refer to the implementation with binary tries. For ordered alphabets, the bound $O(|w|\log |\Sigma|)$ extends also to the list implementation of the symbol-based downward labels. To summarize our discussion, the multiplicative factor $t$ appearing in Theorem 2 and in on-line search is a logarithm, the argument of which can be made to be either $|\Sigma|$ or $n$. Clearly, we would choose $|\Sigma|$ when $\Sigma$ is finite.
5 Building suffix trees in parallel

We address now the parallel construction of the suffix tree $T_x$ associated with input string $x$. We adopt the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM) model of computation described in the first Chapter of the book. We use $n$ processors which can simultaneously read from and write to a common memory with $\Theta(n^2)$ locations. When several processors attempt to write simultaneously to the same memory location, one of them succeeds but we do not know in advance which. Note that an algorithm takes care in general of initializing the memory it uses. In this particular case, however, we will show that a memory location is read by some processor only after that processor attempted to write to it. Thus, we do not need to initialize this space. The overall processors \times time cost of our algorithm is $O(n \log n)$, which is optimal when $|\Sigma|$ is of the same order of magnitude as $n$. It is left as an exercise to show that the space can be reduced to $O(n^{1+\epsilon})$, for any chosen $0 < \epsilon \leq 1$, with a corresponding slow-down proportional to $1/\epsilon$.

From now on, we will assume w.l.o.g. that $n - 1$ 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. Our idea is to start with a tree $D_x$ which consists simply of a root node with $n$ children, corresponding to the first $n$ suffixes of $x\#$, and then produce $\log n$ consecutive refinements of $D_x$ such that the last such refinement coincides with $T_x$ up to a reversal of the direction of all edges. The edges in $D_x$ and each subsequent refinement point from each node to its parent. Throughout, information is stored into the nodes and leaves. Specifically, each leaf or internal node of a refinement of $D_x$ is labeled with the descriptor of some substring of $x\#$ having starting positions in $[1,n]$. We adopt pairs in the form $(i,l)$, where $i$ is a position and $l$ is a length, as descriptors. Thus, the root of $D_x$ is the locus of the empty word. The root has $n$ sons, each one being the locus of a distinct suffix of $x$.

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 $i$-th position of $x\#$, $i = 1, 2, \ldots, n$. Our computation consists of a preprocessing phase followed by a processing phase. They are described next.

5.1 Preprocessing

The preprocessing consists of partitioning the substrings of $x\#$ of length $2^q (q = 0, 1, \ldots, \log n)$ into equivalence classes, in such a way that substrings that are identical end in the same class. For this, each processor is assigned $\log n + 1$ cells of the common memory. The segment assigned to $p_i$ is called $ID_i$. By the end of the preprocessing, $ID_i[q]$ ($i = 1, 2, \ldots; q = 0, 1, \ldots, \log n$) contains (the first component of) a descriptor for the substring of $x\#$ of length $2^q$ which starts at position $i$ in $x\#$, in such a way that all the occurrences of the same substring of $x$ get the same descriptor. For convenience, we
extend the notion of $ID$ to all positions $i > n$ through the convention: $ID_i[q] = n + 1$ for $i > n$. We will use a bulletin board (BB) of $n \times (n + 1)$ locations in the common memory. According to our convention, all processors can simultaneously attempt to write to BB and simultaneously read from it. In the following, we call $\text{winner}(i)$ the index of the processor which succeeds in writing to the location of the common memory attempted by $p_i$.

The initializations are as follows. In parallel, all processors initialize their ID arrays filling them with zeroes. Next, the processors partition themselves into equivalence classes based on the symbol of $\Sigma$ faced by each. Treating symbols as integers, processors that face the same symbol attempt to write their serial number in the same location of BB. Thus, if $x_i = s \in \Sigma$, processor $p_i$ attempts to write $i$ in $BB[1, s]$. Through a second reading from the same location, $p_i$ reads $j = \text{winner}(i)$ and sets $ID_i[0] \leftarrow j$. Thus $(j, 1)$ becomes the descriptor for every occurrence of symbol $s$.

We now describe iteration $q$, $q = 1, 2, \ldots , \log n$, which is also performed synchronously by all processors. Processor $p_i$, $i = 1, 2, \ldots , n$ first grabs $ID_{i+q[q]}$ and then attempts to write $i$ in $BB[ID_i[q], ID_{i+2q[q]}]$. Finally, $p_i$ sets: $ID_i[q + 1] \leftarrow \text{winner}(i), i = 1, 2, \ldots , n$. Note that, since no two $n$-symbol substrings of $x#$ are identical, $p_i$ ($i = 1, 2, \ldots , n$) must be writing its own number into $ID_i[\log n]$ at the end of the computation. Note that a processor reads from a location of BB only immediately after attempting to write to that location. Our discussion of preprocessing establishes the following theorem.

**Theorem 3** There is an algorithm to compute the ID tables in $O(\log n)$ time and $\Theta(n^2)$ space with $n$ processors in a CRCW.

### 5.2 Structuring $D_x$

We need some conventions regarding the allocation of $D_x$ and of its subsequent refinements. For this purpose, we assign to each processor another segment of the common memory, also consisting of $\log n + 1$ cells. The segment assigned to $p_i$ is called $\text{NODE}_i$. Like the ID tables, $\text{NODE}_i$ is made empty by $p_i$ at the beginning. Our final construction takes as input the string $x#$, a location of the common memory called $\text{ROOT}$, and the arrays $ID_i[q]; (i = 1, 2, \ldots , n, q = 0, 1, \ldots , \log n)$, and computes the entries of the arrays $\text{NODE}_i[q]$ ($i = 1, 2, \ldots , n, q = 0, 1, \ldots , \log n$). By the end of the computation, if, for some value of $q \leq \log n$, $\text{NODE}_i[q]$ is not empty, then it represents a node $\mu$ created with the $k$th refinement of $D_x$, where $k = \log n - q$, with the following format: the field $\text{NODE}_i[q].\text{LABEL}$ represents $\text{label}($$\mu$), and the field $\text{NODE}_i[q].\text{PARENT}$ points to the $\text{NODE}$ location of Father[$\mu$]. The initialization consists of setting:

\[ \text{NODE}_i[\log n].\text{PARENT} \leftarrow \text{address}(\text{ROOT}); \]
\[ \text{NODE}_i[\log n].\text{LABEL} \leftarrow (ID_i[\log n], n) \]
Hence $NODE_i[\log n]$ becomes the locus of $suf_i$. Note that $NODE_i[\log n]$ stores the leaf labeled $(i, n)$ and thus is nonempty for $i = 1, 2, ..., n$.

To familiarize with the $NODE$ tables, we consider the process that produces the first refinement of $D_x$. Essentially, we want to partition the edges of $D_x$ into equivalence classes, putting edges labeled with the same first $n/2$ symbols in the same class. For every such class, we want to funnel all edges in that class through a new internal node, which is displaced $n/2$ symbols from the root.

We do this as follows. Assume one row known to all processors, say, row $r$ of $BB$ is assigned to $ROOT$. Then, processors facing the same label in $ID[\log n - 1]$ attempt to write their serial number in the same location of this row of $BB$. Specifically, if $ID_i[\log n - 1] = k$, processor $p_i$ attempts to write $i$ in $BB[r, k]$. Through a second reading from the same location, $p_i$ reads $j = winner(i)$. This elects $NODE_j[\log n - 1]$ to be the locus in the new tree of strings having label $(j, n/2)$. Processor $p_j$ copies this pair into $NODE_j[\log n - 1].LABEL$ and sets a pointer to $ROOT$ in $NODE_j[\log n - 1].LABEL$.

For all $i$ such that $winner(i) = j$, processor $p_i$ sets:

\[
\begin{align*}
NODE_i[\log n - 1].PARENT & \leftarrow \text{address}(NODE_j[\log n - 1]) \\
NODE_i[\log n - 1].LABEL & \leftarrow ID_{i+n/2}[\log n - 1]
\end{align*}
\]

We shall see shortly that some additional details need to be fixed before this refinement of $D_x$ can be deemed viable. For instance, nodes having a single child must be forbidden in any of the refinements. This means that, whenever a node $\mu$ is created that has no siblings, then the pointer from $\text{Father}[\mu]$ must be removed and copied back into $\mu$. Taking care of this problem is not difficult. A more serious problem is the following one. Recall that we started out with the processors sitting on locations of the $NODE$ arrays that correspond to the leaves of $D_x$. As a result of the first refinement, we have now internal nodes other than the root. In order to proceed with our scheme, we need to equip these internal nodes each with its own processor. Since we avoid the formation of unary nodes we will need no more than $2n$ processors at any point of our computation. However, there is no way to predict which $NODE$ locations will host the newly inserted nodes, and there are $\Theta(n\log n)$ such locations. Thus, the main difficulty is designing a scheme that assigns dynamically processors to nodes in such a way that every node gets its processor.

5.3 Refining $D_x$

We concentrate now on the task of producing $\log n$ consecutive refinements of $D_x = D^{(\log n)}$. The $q$-th such refinement is denoted by $D^{(\log n-q)}$. The last refinement $D^{(6)}$ is identical to $T_x$ except for the edge directions, which are reversed.

We will define our sequence of refinements by specifying how $D^{(\log n-q)}$ is obtained from $D^{(\log n-q+1)}$, for $q = 1, 2, ..., \log n$. Three preliminary notions are needed in order to proceed.
A nest is any set formed by all children of some node in $D^{(k)}$. Let $(i, l)$ and $(j, f)$ be the labels of two nodes in some nest of $D^{(k)}$. An integer $t$, $0 < t \leq \min[l, f]$, is a refiner for $(i, l)$ and $(j, f)$ iff $x^#[i, i + t - 1] = x^#[j, j + t - 1]$. A nest of $D^{(k)}$ is refinable if $2^{k-1}$ is a refiner for every pair of labels of nodes in the nest.

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

(i) $D^{(k)}$ is a rooted tree with $n$ leaves and no unary nodes;

(ii) Each node of $D^{(k)}$ is labeled with a descriptor of some substring of $x$; each leaf is labeled, in addition, with a distinct position of $x$; the concatenation of the labels on the path from the root to leaf $j$ describes $suf_j$.

(iii) No pair of labels of nodes in a same nest of $D^{(k)}$ admits a refiner of size $2^k$.

Observe that condition($\log n$) 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. Let $(i_1, l_1), (i_2, l_2), \ldots, (i_m, l_m)$ be the set of all labels in some nest of $D^{(k)}$. Let $\nu$ 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 $\nu$, as follows. With the child node labeled $(i_j, l_j)$ associate the contracted label $ID_{[k - 1]}$, $j = 1, 2, \ldots, m$. Now partition the children of $\nu$ into equivalence classes, putting in the same class all nodes with the same contracted label. For each non-singleton class which results, perform the following three operations.

(1) Create a new parent node $\mu$ for the nodes in that class, and make $\mu$ a son of $\nu$.

(2) Set the LABEL of $\mu$ to $(i, 2^{(k-1)})$, where $i$ is the contracted label of all nodes in the class.

(3) Consider each child of $\mu$. 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 $\mu$ as above was created for the nodes in $C$ during STEP 1. Make $\mu$ a child of the parent of $\nu$ and set the LABEL of $\mu$ to $(i, l + 2^{k-1})$, where $(i, l)$ is the label of $\nu$.

The following lemma shows that our definition of the series of refinements $D^{(k)}$ is unambiguous.
Lemma 6 The synchronous application of Steps 1 and 2 to all nests of $D^{(k)}$ produces a tree that meets condition $(k - 1)$.

Proof: Properties (ii-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.

Since $|\$| $\cup \Sigma > 1$, then the nest of the children of the root cannot end in a singleton class for any $k > 0$. Thus for any parent node $\nu$ of a nest of $D^{(k)}$ involved in STEP 2, Father[\nu] is defined. By condition $(k)$, node $\nu$ has more than one child, and so does Father[\nu]. Let $\bar{D}^{(k)}$ be the structure resulting from application of Step 1 to $D^{(k)}$.

If, in $D^{(k)}$, the nest of Father[\nu] is not refinable, then $\nu$ is a node of $D^{(k-1)}$, and $\nu$ may be the only unary node in $\bar{D}^{(k)}$ between any child of $\nu$ in $D^{(k)}$ and the parent of $\nu$ in $D^{(k)}$. Node $\nu$ is removed in STEP 2, unless $\nu$ 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 $\nu$ and that of Father[\nu] are refinable. We claim that, in $\bar{D}^{(k)}$, either the parent of $\nu$ has not changed and it is a branching node, or it has changed but still is a branching node. Indeed, by definition of $D^{(k)}$, neither the nest of $\nu$ nor that of Father[\nu] can be refined into only one singleton equivalence class. Thus, by the end of STEP 1, the following alternatives are left.

1. The Father of $\nu$ in $\bar{D}^{(k)}$ is identical to Father[\nu] in $D^{(k)}$. Since the nest of Father[\nu] could not have been refined into only one singleton class, then Father[\nu] must be a branching node in $D^{(k-1)}$. Thus this case reduces to that where the nest of Father[\nu] is not refinable.

2. The parent of $\nu$ in $\bar{D}^{(k)}$ is not the parent of $\nu$ in $D^{(k)}$. Then Father[\nu] in $\bar{D}^{(k)}$ is a branching node, and also a node of $D^{(k-1)}$. If $\nu$ is a branching node in $\bar{D}^{(k)}$, then there is no unary node between $\nu$ and Father[\nu] in $\bar{D}^{(k)}$, and the same holds true between any node in the nest of $\nu$ and $\nu$. If $\nu$ is an unary node in $\bar{D}^{(k)}$, then the unique child of $\nu$ is a branching node. Since the current parent of $\nu$ is also a branching node by hypothesis, then removing $\nu$ in STEP 2 eliminates the only unary node existing on the path from any node in the nest of $\nu$ to the closest branching ancestor of that node. □

If the nest of $D^{(k)}$ rooted at $\nu$ 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 contracted label for that node using its LABEL field and the ID tables. Next, the processors use the row of $BB$ assigned to the nest and the contracted labels to partition themselves into equivalence classes: each processor in the nest whose contracted label is $i$ competes to write the address of its node in the $i$th 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 create now the new
parent nodes, label them with their contracted 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 label prescribed by STEP 2.

We conclude that once each node of $D^{(k)}$ is assigned a distinct processor, $D^{(k-1)}$ can be produced in constant time. The difficulty, however, is how to assign $n$ constant time additional processors to the nodes created anew in $D^{(k-1)}$. It turns out that bringing fewer processors into the game leads to a crisp processor (re-)assignment strategy. The basic idea is to perform the manipulations of Steps 1-2 using $m-1$ processors, rather than $m$ for a nest of $m$ nodes. The only prerequisite for this is that 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. This can be used also to assign the rows of $BB$ to the nodes of $D^{(k)}$: simply assign the $i$-th 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. In summary, we stipulate the following

**Invariant 2** In any refinement of $D_x$, if a node other than ROOT has $m$ children, then precisely $m-1$ of the children have been assigned a processor. Moreover, each one of the $m-1$ processors knows the address of the unique sibling without a processor.

For any given value of $k$, let a **legal** assignment of processors to the nodes of $D^{(k)}$ be an assignment that enjoys Invariant 2.

**Lemma 7** 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 re-allocates the processors in each nest of $D^{(k)}$ on the nodes of $D^{(k)}$. We show then that our policy leads to a legal assignment for $D^{(k-1)}$.

Let then $\nu$ be the parent of a nest of $D^{(k)}$. A node to which a processor has been assigned will be called *pebbled*. By hypothesis, all but one of the children of $\nu$ are pebbled. Also, all children of $\nu$ are nodes of $D^{(k)}$. In the general case, some of the children of $\nu$ in $D^{(k)}$ are still children of $\nu$ in $D^{(k)}$, 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 $\mu_r$ of $D^{(k)}$ such that all children of $\mu_r$ 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 $\nu$ is removed, then its pebble is passed down to the (unique) son $\mu$ of $\nu$ in $D^{(k)}$. 

15
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 $\nu$ is removed in the transition from $D^{(k)}$ to $D^{(k-1)}$, then the unique son $\mu$ of $\nu$ in $D^{(k)}$ is unpebbled in $D^{(k)}$. 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 $\nu$ 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 $\nu$ of $D^{(k)}$ into nodes of $D^{(k-1)}$, as follows. If node $\mu$ is in the nest of $\nu$ and also in $D^{(k-1)}$ then set $\mu' = f(\mu) = \mu$; if instead $\mu$ is not in $D^{(k-1)}$, let $\mu' = f(\mu)$ be the (unique) son of $\mu$ in $D^{(k)}$.

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

In Case 2, node $\nu$ is in $D^{(k-1)}$. Any node $\mu$ in the nest of $\nu$ is in $D^{(k)}$. At the end of STEP 2, the pebble of node $\mu$ will go untouched unless $\mu$ 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 $\nu$ 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 $\mu$ which was in the nest of $\nu$ in $D^{(k)}$, node $\mu'$ is pebbled iff $\mu$ was pebbled at the end of STEP 1, which concludes the proof. $\square$

### 5.4 Reversing the edges

In order to transform $D^{(0)}$ into a suffix tree we need only to reverse the direction of all edges. For simplicity, we retain the format according to which edge labels are assigned to the child, rather than to the father node in an edge. We must still add to each node a branching label of the kind discussed in Section 4. As seen in that section, there are various ways of implementing these labels. We will limit our description to the trie implementations of symbol-based branching labels and the array implementation of ID-based branching labels, since all the others can be derived from one of these two quite easily.

To implement symbol-based labels with tries, we need to replace each original internal node of $D^{(0)}$ with a binary trie indexing to a suitable subset of $\Sigma$. This transformation can be obtained in $O(\log |\Sigma|)$ 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 $\log |\Sigma|$ further refinements of $D^{(0)}$, for which the ID tables are not needed. In fact, the best descriptor for a string of $\log |\Sigma|$ 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 label $(i,l)$ will use symbol $x_i$ in this process.
Whenever a new branching node $\nu$ is created, one of the processors in the current nest of $\nu$ climbs to $\mu = \text{Father}[\nu]$ and assigns the appropriate branching label to $\mu$. At the end, the processors assign branching labels to the ultimate fathers of the nodes in the nest.

For the array implementation of $1D$-based branching labels, we assign a vector of size $n$, called $\text{OUT}_{\nu}$, to each node $\nu$ of $D^{(0)}$. The vector $\text{OUT}_{\nu}$ stores the branching label from $\nu$ as follows. If $\mu$ is a son of $\nu$ and the label of $\mu$ is $(i,l)$, a pointer to $\mu$ is stored in $\text{OUT}_{\nu}[ID_{\nu}]$. 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 $\text{OUT}$ vectors for all refinements $D^{(k)}$ would become $\Theta(n^2 \log n)$. Observe that, since $n$ processors cannot initialize $\Theta(n^2)$ space in $O(\log n)$ time, the final collection of $\text{OUT}$ 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 $\text{OUT}$ vector of the parent node into a linked list. This task is accomplished trivially in extra $O(\log n)$ time, using prefix computation.

**Theorem 4** The suffix tree in compact form for a string of $n$ symbols can be built in $O(\log n)$ steps by $n$ processors in a CRCW-PRAM, using $O(n^2)$ auxiliary space without need for initialization.

**Proof:** The claim is an obvious consequence of Theorem 3, lemmas 6 and 7 and the discussion above. □

As we see shortly, $T_x$ alone is not enough to carry out on-line string searching in parallel. For this, we shall need the entire series of $D^{(k)}$'s as implemented by $\text{OUT}$ vectors.

### 6 Parallel on-line search

Assume that, in the course of the construction of the suffix tree associated with string $x:\#$, we saved the following entities: (1) The $\log n$ bulletin boards used in the construction of the $1D$ tables. (2) All the intermediate trees $D^{(k)}$, $k = \log n, \ldots, 0$, each implemented by the vectors $\text{OUT}_{\nu}$, defined the previous section. Note that this assumption presupposes $O(n^2 \log n)$ space. We show that, with this information available, $m$ processors can answer in $O(\log m)$ steps whether a pattern $y = y_1y_2\ldots y_m$ occurs in $x$. Formally, we list the following

**Theorem 5** Let $x$ be a string of $n$ symbols. There is an $O(n \log n)$-work preprocessing of $x$ such that, for any subsequently specified pattern $y = y_1y_2\ldots y_m$, $m$ processors in a CRCW can find whether $y$ occurs in $x$ in time $O(\log m)$. 

17
Proof: We give an explicit construction that meets the claim, assuming conditions (1) and (2) above were satisfied during preprocessing. We perform our on-line search in three main steps, as follows.

**Step 1.** Recall that 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 label for the substring $x_i, \ldots, x_{i+2^q-1}$, such that $ID_i[q] = ID_j[q]$ if $x_i, \ldots, x_{i+2^q-1} = x_j, \ldots, x_{j+2^q-1}$. The first step of the on-line search for $y$ consists of labeling in a similar way some of the substrings in the pattern $y$. For $q = 0, \ldots, \log m$, the substrings we assign labels to are all substrings whose length is $2^q$ and starting at every position $i$ such that $i$ is a multiple of $2^q$ and $i + 2^q \leq m$. These new labels are stored in the vectors $PID_i[q]$, so that $PID_i[q]$ stores the label of the substring $y_i, \ldots, y_{i+2^q-1}$. $PID$ labels are assigned in such a way that whenever two substrings of length $2^q$, one in $y$ and the other in $x$#, are equal then their labels are equal too. For this, we follow a paradigm similar to that used in deriving the $ID$ labels, but we do not compute the $PID$ labels from scratch. Instead, we just copy appropriate entries of the bulletin boards ($BB$s) used in deriving the $ID$ labels. Since the $BB$ tables were not initialized, then every time we copy an entry of a $BB$ table, we need to check the consistency of such an entry with the corresponding entry of an $ID$ table. Should we find no correspondence at any step of this process, we can conclude that there is a substring of the patterns that never occurs in the text, whence the answer to the query is NO.

**Step 2.** Let $PID_1[\log m]$ (that is, the name of the prefix of $y$ whose length is $2^{\log m}$) be $h$. Observe that if none of $PID_1[\log m]$ is equal to $h$ then the prefix of $y$ whose length is $2^{\log m}$ does not occur in $x$. We conclude that $y$ does not occur in $x$ whence the answer to the query is NO.

Suppose $h = PID_i[\log m]$ for some $1 \leq i \leq n - 1$. We check whether $NODE_{E_1}[\log m]$ appears in $D_{\log n-1}$. Note that $NODE_{E_1}[\log m]$ will not appear in $D_{\log n-1}$ if and only if all the substrings of $x$ whose prefix of length $2^{\log m}$ is the same as the prefix of $y$ have also the same prefix of length $2^{\log m+1}$. If $NODE_{E_1}[\log m]$ appears in $D_{\log n-1}$ then we are guaranteed that it will appear also in $D_{\log m}$ and we proceed to Step 3. In fact, all the refinements $D_{\log n-1}, \ldots, D_{\log m}$ deal only with substrings whose length is greater than $2^{\log m}$. Otherwise, i.e., $NODE_{E_1}[\log m]$ does not appear in $D_{\log n-1}$, we check whether $y$ is equal to $x_h, \ldots, x_{h+n-1}$ symbol by symbol. 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 $T_x$ such that $y$ is a prefix of the string having $v$ as its locus (if such a node exists). For this, we use the vectors $PID_i[q]$, of Step 1 and the $D_{\theta}$ trees, $q = \log m - 1, \ldots, 0$, of the preprocessing. Node $v$ is found thru a "binary search" of $\log m$ iterations, as follows.

*Iteration* $q$ ($q = \log m - 1, \ldots, 0$). Let $v$ and $y'$ be the input parameters of iteration $q$. (For iteration $\log m - 1$, $v = NODE_{E_1}[\log m]$ and $y'$ is the suffix of $y$ starting at position $2^{\log m} + 1$.) The invariant property satisfied in all the iterations is that $v$ is a node in
$D^{(q+1)}$ and $y'$ is a substring whose length is less than $2^q+1$. Our goal is to check whether $y'$ follows an occurrence of $W(\nu)$. We work on $D^{(q)}$. There are two possibilities:

(Possibility 1) The node $\nu$ appears in $D^{(q)}$. Possibility 1 has two subpossibilities.

(Possibility 1.1) $2^q$ is larger than the length of $y'$. In this case we do nothing and the input parameters of the present iteration become the input parameters of the next iteration. (Possibility 1.2) $2^q$ is less than or equal to the length of $y'$. Assume that $y'$ starts at position $j$ of $y$ and $b$ is the value stored in $PID_j[q]$. If the entry $OUT_{\nu}[b]$ is empty then $y$ does not occur in $x$. Otherwise, the input parameters of the next iteration will be the suffix of $y'$ starting at position $2^q + 1$ and the node pointed to by $OUT_{\nu}[b]$.

(Possibility 2) The node $\nu$ does not appear in $D^{(q)}$. This means that $\nu$ had only one son in $D^{(q+1)}$ and so it was omitted from $D^{(q)}$ (in Step 2 of refining $D^{(q+1)}$). Let $\mu$ be the single son of $\nu$ in $D^{(q+1)}$. Possibility 2 has two subpossibilities. (Possibility 2.1) $2^q$ is larger than the length of $y'$. Assume that the $LABEL$ of $\mu$ in $D^{(q)}$ is $(i, l)$. In this case $y'$ occurs in $x$ if and only if $y'$ is a prefix of $x_{i+i-2i+1}, \ldots, x_{i+l}$. We check this letter by letter in $\log m$ time using $m/\log m$ processors. (Possibility 2.2) $2^q$ is less or equal to the length of $y'$. We compare $ID_{i+i-2i+1}[q]$ (the unique name of $x_{i+i-2i+1}, \ldots, x_{i+l}$) to the unique name of the prefix of $y'$ whose length is $2^q$. If these names are different then $y$ does not occur in $x$. Otherwise, the input parameters of the next iteration will be the suffix of $y'$ starting at position $2^q + 1$ and the node $\mu$.

As a final remark, observe that we did not initialize the vectors $OUT_{\nu}$, therefore it could be that we will get a wrong positive answer. To avoid mistakes, every time we get a positive answer we need to explicitly check whether $y$ really appears in $x$ at the position given in the answer. This can be done in $\log m$ time using $m/\log m$ processors as a last step. \(\square\)

7 Exhaustive on-line searches

Given $T_r$ in compact form and Fact 2 of Section 2, one can find, for any pattern $y$, all the occurrences of any substring $w$ of $y$ in $x$ in serial time $O(|w| + l)$, $l$ being the total number of occurrences of $w$ in $x$. This application is a special case of the following broader problem. Assume we are given a set of strings $W$ upon which we want to perform many substring queries, as follows. In each query, we specify arbitrarily a substring $w'$ of some string $w$ in $W$ (possibly, $w' = w$) as the pattern, ad also a set $W' = \{\bar{w}_1, \bar{w}_2, \ldots, \bar{w}_t\}$ of textstrings, where each $\bar{w}$ is a string from $W$ or a substring of one such string. The result of the query is the set of all the occurrences of $w'$ in $W'$. The quantity $\bar{n} = \sum_{k=1}^{t} |\bar{w}_k| + |w'|$ is the size of the query. This kind of queries arise naturally in sequence data banks, and they have obvious (off-line) serial solution taking time linear in $\bar{n}$. We investigate now their efficient on-line parallel implementation.

It can be proved that the strings in a data bank can be preprocessed once and for all in such a way that any subsequent substring query on the bank takes constant time on a CRCW PRAM with a number of processors linear in the size of the query. Preprocessing
a string $x$ costs $O(\log |x|)$ CRCW-PRAM steps and $O(|x| \log |x|)$ total work and space. Note that the methods used in off-line parallel searches depend crucially on the specific patterns being considered and thus do not support instantaneous substring queries. For space limitations, we will describe only part of the method, suitable for a restricted class of inputs. But our discussion will suffice to display an interesting fact, namely, that assuming an arbitrary order on the input alphabet may lead to efficient solutions to problems on strings to which the notion of alphabet order is totally extraneous.

Let then the alphabet $\Sigma$ be ordered according to the linear relation $\prec$. This order induces a lexicographic order on $\Sigma^*$, which we also denote by $\prec$. Given two words $u$ and $v$, we write $u \ll v$ or $v \gg u$ to denote that there are two symbols $a$ and $a'$ with $a < a'$, and a word $z \in \Sigma^*$ such that $za$ is a prefix of $u$ and $za'$ is a prefix of $v$. Thus, $u < v$ iff either $u \ll v$ or $v \gg u$ is a prefix of $v$.

**Fact 3** Let $u \ll v$. Then, for any $w$ and $z$ in $\Sigma^*$, we have $uw \ll vz$.

If $x = vwy$, then the integer $1 + |v|$, where $|v|$ is the length of $v$ is the (starting) position in $x$ of the substring $w$ of $x$. Let $I = [i,j]$ be an interval of positions of a string $x$. We say that a substring $w$ of $x$ begins in $I$ if $I$ contains the starting position of $w$, and that it ends in $I$ if $I$ contains the position of the last symbol of $w$.

We recall few notions from the introductory chapter. A string $w$ is primitive if it is not a power of another string (i.e., writing $w = v^k$ implies $k = 1$). A primitive string $w$ is a period of another string $z$ if $z = w^cw'$ for some integer $c > 0$ and $w'$ a possibly empty prefix of $w$. A string $z$ is periodic if $z$ has a period $w$ such that $|w| \leq |z|/2$. It is a well-known fact of combinatorics on words that a string can be periodic in only one period. We refer to the shortest period of a string as the period of that string.

A string $w$ is a square if it can be put in the form $vv$ in terms of a primitive string $v$ ($v$ is the root of the square). A string is square-free if none of its substrings is a square. Our implementation of fast substring queries will be discussed under the very restrictive assumption that all strings we handle are square-free. In the general method, this assumption can be waived without any penalty in efficiency.

We can explain the basic criterion used in our construction in terms of the standard, single-pattern string searching problem. Let then $y$ s.t. $|y| \geq 4$ be this pattern and $x$ a text string, as in Figure 4. Consider the ordered set $S$ of all positioned substrings of $y$ having length $c = 2(\log |y|| - 2)$, and let $(i,s)$ be the one such substring such that $s$ is a lexicographic minimum in $S$ and $i$ the smallest starting position of $s$ in $y$. Substring $(i,s)$ is called the seed of $y$. Pattern $y$ is left-seeded if $i < c$, right-seeded if $i > |y| - 2c + 1$, balanced in all other cases.

Let now the positions of $x$ be also partitioned into cells of equal size $c = 2(\log |y|| - 2)$, and assume that there is at least one occurrence of $y$ in $x$, starting in some cell $B$. In principle, every position of $B$ is equally qualified as a candidate starting position for an occurrence of $y$. However, the same is not true for the implied occurrences of the seed of $y$. This seed will start in a cell $B'$ that is either $B$ itself or a close neighbor of $B$. 

20
Consider the set of all substrings of $x$ which start in $B'$ and have length $|s|$. It is not difficult to see then that the one such positioned substring corresponding to $(i, s)$ has the property of being a lexicographic minimum among all such substrings originating in $B'$ and to its right, or originating in $B'$ and to its left, or both, depending on whether $y$ is left-, right-seeded, or balanced. Once we have a candidate position for $s$ in $B'$, it is possible to check in constant time with $s$ processors whether this actually represents an occurrence of $y$, since $|y| \leq 8|s|$. The problem is thus to identify such a candidate position. Note that, although we know that the seed of, say, a left-seeded pattern must be lexicographically least with respect to all substrings of equal length that begin in $B'$ and to its right, there might be up to $|s| = |B'|$ substrings with this property. Even if we were given the starting positions of all such substrings, checking all of them simultaneously might require $|s|^2$ processors.

Throughout the rest of this discussion, we concentrate on the management of left-seeded patterns, but it shall be apparent that the case of right-seeded patterns is handled
7.1 Lexicographic lists

Let \( B = [h, h+m] \), where \( m \leq n/2 \) and \( h \leq (n-2m+1) \), be a cell of size \( m \) on out string \( x (|x| = n) \). A stub of \( B \) is any positioned substring \( (i, z) \) of \( x \) of length \( |z| = m \) and \( i \in B \). Stub \( (i, z) \) is a left stub of \( B \) if \( i = h + m \) or, for any other stub \( (i', z') \) of \( B \) with \( i < i' \), we have \( z \leq z' \). We use \( L(B) = \{(i_1, z_1), (i_2, z_2), ..., (i_k, z_k)\} \) to denote the sequence of left stubs of \( B \), and call the ordered sequence \( \{i_1, i_2, ..., i_k\} \), where \( i_1 < i_2 < ... < i_k \), the left list of \( B \).

As an example, let the substring of \( x \) in block \( B \) be \( eacaccdlacdacdacdlllh \), as in Figure 5, and assume for simplicity that the positions of \( x \) falling within \( B \) be in \([1,22] \).

![Figure 5: Left stubs in a block](image)

We have 8 left stubs in \( B \), beginning with the rightmost such stub \( (22, z_6) = f... \). Since \( h \) and \( l \) are both larger than \( f \), the next left stub is \( (17, z_7) = dlllh f... \). We immediately have \( (16, z_6) = cdlllh f... \) and \( (15, z_5) = acdlllh f... \). Since \( d \) and \( c \) are both larger than \( a \), there will not be a left stub until \( (12, z_4) = acdacdlllh f... \). We similarly have \( (9, z_3) = acdacdacdlllh f... \). Finally, we have \( (4, z_2) = acdlaacdadaacdlla f... \) and \( (2, z_1) = acacdaacdadaacdlllh f... \). Note that the prefix of \( z_1 \) of length 2 = \( i_2 - i_1 \) matches the corresponding prefix of \( z_2 \). Similarly, the prefix of \( z_3 \) of length 3 = \( i_4 - i_3 \) matches a prefix of \( z_4 \), and the prefix of \( z_4 \) of length 3 = \( i_5 - i_4 \) matches a prefix of \( z_5 \). We say that \( z_1 \) and \( z_2 \) are in a run, and so are \( z_3, z_4 \) and \( z_5 \). Obviously, there can be no runs in a square-free string.
Lemma 8 Assume that \( x \) is square-free, and let \((i, z)\) and \((j, z')\) be two consecutive left stubs from \( S(B) \). Then, \( i < j \) implies \( z' \ll z' \), where \( z' \) and \( z' \) are the prefixes of \( z \) and \( z' \) of length \( |j - i| \).

Proof: Straightforward. \( \Box \)

Let now \( \mathcal{L}(B) = (i_1, z_1), (i_2, z_2), \ldots, (i_k, z_k) \) be the ordered sequence of left stubs in \( S(B) \). If \( k = 1 \), then define \( \mathcal{I}(B) \) as the singleton set containing only \((i_1, z_1)\). Assume henceforth \( k > 1 \). For \( f = 1, 2, \ldots, k - 1 \), let \( l_f \) be the prefix of \( z_f \) such that \( |l_f| = i_{f+1} - i_f \). We use \( \mathcal{I}(B) \) to denote the ordered sequence \((i_1, l_1), (i_2, l_2), \ldots, (i_{k-1}, l_{k-1})\). With each \((i_f, l_f) \in \mathcal{I}(B)\), we associate its shadow \((i_{f+1}, l'_f)\), where \( l'_f \) is the prefix of \( z_{f+1} \) having the same length as \( l_f \). The ordered sequence of shadows of the members of \( \mathcal{I}(B) \) is denoted by \( \mathcal{I}'(B) \). By construction, we have that \( l_f \leq l'_f \) for each \( f \) in \([1, k - 1]\). If, in addition, \( x \) is square-free, then Lemma 8 ensures that \( l_f \ll l'_f \) for each \( f \) in \([1, k - 1]\). We now use the elements of \( \mathcal{I}(B) \cup \mathcal{I}'(B) \) to construct the ordered sequence \( \mathcal{I}(B) = (i_1, l_1), (i_2, l_2), \ldots, (i_k, l_k) \) defined as follows (cf. Figure 6).

First, we set \( \tilde{l}_1 = l_1 \) and \( \tilde{l}_k = l_{k-1} \). Next, for \( 1 < f < k \), we set \( \tilde{l}_f = l'_{f-1} \) if \( i_{f+1} - i_f < i_f - i_{f-1} \), and \( \tilde{l}_f = l_f \) otherwise. Sequence \( \mathcal{I}(B) \) plays an important role in our constructions, due to the following lemmas.

Lemma 9 If \( x \) is square-free, then the word terms in \( \mathcal{I}(B) \) form a lexicographically strictly increasing sequence.

Proof: We prove that, for \( k > 1 \), we must have \( \tilde{l}_1 \ll \tilde{l}_2 \ll \ldots \ll \tilde{l}_k \). This is easily seen by induction. By Lemma 8, \( \tilde{l}_1 = l_1 \ll l'_1 \). By our definition of \( \tilde{l}_2 \), we have \( |\tilde{l}_2| \geq |l'_1| \), i.e., \( l'_1 \) is a prefix of \( l_2 \). By Fact 2, we get then that \( \tilde{l}_1 \ll \tilde{l}_2 \). Assuming now that the claim holds for all values of \( f \) up to \( f = h < k \), the same argument leads to establish that \( \tilde{l}_h \ll \tilde{l}_{h+1} \).

\( \Box \)

Lemma 10 The sum of the lengths of the word terms in \( \mathcal{I}(B) \) is bounded above by \( 4|B| \).

Proof: Each \( \tilde{l} \) derives its length either from a distinct \( l \) or from a distinct \( l' \). Since the \( l' \)'s do not mutually overlap, then their total length is bounded by \( 2m \), and the same is true of the \( l \)'s. \( \Box \)

Lemmas 9 and 10 justify our interest in \( \mathcal{I}(B) \). In fact, Lemma 9 states that if \( x \) is square-free, then there is at most one member \((\tilde{i}, \tilde{l})\) of \( \mathcal{I}(B) \) such that \( \tilde{l} \) is a prefix of seed \( w \). Note that this is not true for the elements of \( \mathcal{I}(B) \), since we may have that, for some \( f \), \( l_{f+1} \) is a prefix of \( l_f \). For example, let \( \ldots adbcadc \) be the last 7 symbols of \( x \) that fall in \( B \). Then \( z_k \) starts with \( c \), while \( z_{k-1} \) and \( z_{k-2} \) start, respectively, with \( adc \) and \( adbcadc \). We have \( l_{k-1} = ad \), which is a prefix of \( l_{k-2} = adbc \). Lemma 10 is a handle to check all these prefixes against \((i, s)\) simultaneously and instantaneously, with \( O(|s|) \) processors.
Observe that given a copy of \( x \), the set \( \mathcal{I}(B) \cup \mathcal{I}'(B) \) is completely specified by the ordered sequence of starting positions of the members of \( \mathcal{I}(B) \), which we called the left list of \( B \). Clearly, the left list of any cell \( B \) enumerates also the starting positions of all elements of \( \mathcal{I}(B) \).

### 7.2 Building lexicographic lists

We show now how a generic square-free string \( w \) is preprocessed. Without loss of generality, we assume \( |w| \) a power of 2. The basic invariant stating that \( w \) is square-free will be called henceforth \( \text{Property 1} \). The preprocessing consists of performing approximately \( \lceil \log |w| \rceil \) stages, as follows. At the beginning of stage \( t \) (\( t = 1, 2, \ldots \)) of the preprocessing the positions of \( w \) are partitioned as earlier into \( |w|/2^{t-1} \) disjoint cells each of size \( 2^{t-1} \). Starting with the first cell \([1, 2^{t-1}]\), we give now all cells consecutive ordinal numbers. For \( t = 1, 2, \ldots \), stage \( t \) handles simultaneously and independently every pair \((B_{od}, B_{od+1})\) of cells such that \( od \) is an odd index. The task of a stage is to build the lexicographic
list relative to every cell $B_d \cup B_{d+1}$, using the lexicographic lists of $B_d$ and $B_{d+1}$). The crucial point is to perform each stage in constant time with $|w| = n$ processors.

We need to make some preliminary arrangements for inter-processor communication. Let our $n$ processors be $p_1, p_2, \ldots, p_n$ where $p_i$ ($i = 1, 2, \ldots, n$) has serial number $i$. The input $w$ is stored into an array of consecutive locations of the common memory, and processor $p_i$ is assigned to the $i$-th symbol $w_i$ of $w$ ($i = 1, 2, \ldots, n$).

The first position of each cell is called the cell head and is assigned a few special memory locations. In our construction, cell heads are used as bulletin boards for sharing information among processors. For example, cell heads are used to record the starting position of the lexicographically least among the stubs that begin in that cell. We use $ls(B)$ to denote this least stub of a cell $B$. Property 0 ensures that $ls(B)$ is unique. Since the partition of the positions of $x$ into cells is rigidly defined for each stage, then the position of any cell head can be computed by any processor in constant time. Throughout our scheme, we need to maintain some invariant conditions that are given next.

**Invariant 3** At the beginning of each stage and for every cell, the starting position of the $ls$ of that cell is stored in the cell head.

We also need that the processors in every cell know the organization of the left list of that cell. The processors use this information in order to compute the sequence $I$ defined earlier. This information is stored according to the invariant properties that are given next. The processor assigned to the starting position of a left stub is called a stub representative.

**Invariant 4** If processor $p$ is assigned to a symbol of a member $(i, l)$ of some sequence $I$, then $p$ knows the serial number of the stub representative of $l$.

**Invariant 5** Every stub representative knows the address of its immediate predecessor in its left list, if the latter exists. Similarly, every stub representative knows the address of its immediate successor in its left list, if the latter exists.

Note that the first element of the left list of a cell is always the starting position of the $ls$ of that cell. Therefore, Property 3 ensures also that the starting position of the first element in the left list of $B$ is stored in the cell head of $B$. Finally, the last element of the left list of a cell is, by construction, always the last position in that cell.

**Theorem 6** Let $w$ be a square-free string, and $(B_d, B_{d+1})$ two consecutive cells in a partition of $w$. Given the left lists of $B_d$ and $B_{d+1}$, the left list of $B = B_d \cup B_{d+1}$ can be produced by a CRCW PRAM with $|B|$ processors in constant time, preserving invariants 3, 4 and 5.
Proof: Consider first the computation \( l_\mathcal{S}(\bar{B}) \). This could be done by straightforward lexicographic pairwise comparisons of appropriate extensions of the current \( l_\mathcal{S}'s \) in adjacent cells. Such extensions consist of the substrings starting with the current \( l_\mathcal{S}'s \) and having size \(|\bar{B}| \). Thus, we compare the extended \( l_\mathcal{S}'s \) of \( B_{od} \) and \( B_{od+1} \). By Property 0, only one of these extensions will survive in the comparison, and the winner coincides with \( l_\mathcal{S}(\bar{B}) \).

Note that in order to know the result of a lexicographic comparison, the processors need to find the leftmost position where the two strings being compared mismatch. A technique to achieve this in constant time was discussed in the introductory chapter.

Our main task, however, is that of combining the two left lists of \( B_{od} \) and \( B_{od+1} \) into the left list for cell \( \bar{B} = B_{od} \cup B_{od+1} \). This is more elaborate than the computation of \( l_\mathcal{S}(\bar{B}) \), but it yields \( l_\mathcal{S}(\bar{B}) \) as a by-product. The basic observation is that, as a consequence of Property 0, the left list of \( B_{od+1} \) is a suffix of the left list of \( \bar{B} \). Thus, the issue is how to identify the prefix of the left list of \( \bar{B} \) to which the left list of \( B_{od+1} \) is to be appended.

Let \( i' \) be the smallest element of the left list of \( B_{od+1} \), and let \( z' \) be the substring of \( w \) having length \( 2^t \) and starting position \( i' \). We use now \( (i_1, \bar{i}_1), (i_2, \bar{i}_2), \ldots, (i_k, \bar{i}_k) \) to denote the sequence \( \bar{l}(B_{od}) \). For \( f = 1, 2, \ldots, k \), let \( \text{shadow}(\bar{i}_f) \) be the prefix of \( z' \) of length \( |\bar{i}_f| \).

Since (cf. Lemma 9) \( \bar{i}_1 \ll \bar{i}_2 \ll \ldots \ll \bar{i}_k \), then precisely one of the following cases must apply.

A) \( \bar{i}_k \ll \text{shadow}(\bar{i}_k) \).
B) \( \bar{i}_1 \gg \text{shadow}(\bar{i}_1) \).
C) There are two consecutive elements \( (i_h, \bar{i}_h) \) and \( (i_h, \bar{i}_{h+1}) \) of \( \bar{l}(B_{od}) \) such that \( \bar{i}_h \ll \text{shadow}(\bar{i}_h) \ll \bar{i}_{h+1} \).
D) There is precisely one element \( (i_h, \bar{i}_h) \) in \( \bar{l}(B_{od}) \) such that \( \bar{i}_h = \text{shadow}(\bar{i}_h) \).

In case (A), the left list for the combined cell \( \bar{B} \) consists of the concatenation of the left lists of \( B_{od} \) and \( B_{od+1} \). If case (B) applies, then the left list of \( \bar{B} \) coincides with the left list of \( B_{od+1} \). In case (C), the left list of \( \bar{B} \) is obtained by appending the left list of \( B_{od+1} \) to the sequence of the first \( h \) elements of the left list of \( B_{od} \). We are thus left with case (D). Let \( i_h \) be the starting position in \( w \) of \( \bar{i}_h \), and \( z \) the substring of length \( 2^t \) of \( w \) having starting position \( i_h \). By Property 0, we must have that either \( z \ll z' \) or \( z \gg z' \).

In the first case, the left list of \( \bar{B} \) is obtained by appending the left list of \( B_{od+1} \) to the sequence of the first \( h \) elements in the left list of \( B_{od} \). In the second case, from \( \bar{i}_{h-1} \ll \bar{i}_h \) and \( \bar{i}_h = \text{shadow}(\bar{i}_h) \) we derive \( \bar{i}_{h-1} \ll z' \), whence the left list of \( \bar{B} \) results from appending the left list of \( B_{od+1} \) to the sequence of the first \( h - 1 \) members of the left list of \( B_{od} \).

This concludes our case analysis. We have to show next that, using the invariants, our \(|\bar{B}| \) processors can perform the computation in constant time.

The preliminary identification of \( \bar{l}(B_{od}) \) is easily performed by the stub representatives: using Invariant 5, each such representative \( p \) can infer the length of its associated word \( \bar{I} \) by comparison of the absolute differences of its own serial number to the serial numbers of its predecessor and successor in the left list, respectively. We want now the processors to compare, in overall constant time, every word \( \bar{I} \) from \( \bar{l}(B_{od}) \) to the prefix of length \(|\bar{I}| \) of \( l_\mathcal{S}(B_{od+1}) \). For every \( \bar{I} \), we need \(|\bar{I}| \) processors for the comparison that involves
\(\bar{l}\) as one of the terms. This cannot be solved by just letting the \(|\bar{l}|\) processors assigned to the symbols of \(\bar{l}\) do the work. In fact, \(\bar{l}\) may overlap with one or more of its successors in \(\bar{I}(B_{cd})\), in which case these successors would simultaneously claim part of the processors of \(\bar{l}\) for comparing their own symbols. One way around this difficulty is to arrange for each \(\bar{l}\) that overlaps its successor to "borrow" at some point the processors needed for the comparison from its own predecessor in \(\bar{I}(B_{cd})\). In fact, our construction of \(\bar{I}(B_{cd})\) guarantees for any \(f\) that, if \(\bar{l}_f\) overlaps with \(\bar{l}_{f+1}\), then \(|\bar{l}_{f-1}| = |\bar{l}_f|\). Since processors are only lent (if needed) to an immediate right successor in the left list, this policy does not generate conflicts. In conclusion, this part of the computation is performed in two substeps. In the first substep, processors assigned to \(\bar{l}\)'s that have no overlap with their successors perform their required lexicographic comparison in a normal way. In the second substep, the representatives of the \(\bar{l}\)'s that overlap send for help from their respective predecessors in the left list, and such predecessors arrange for the comparisons.

The remaining details of a comparison are as follows. Let \(w_d\) be a symbol of a word \(\bar{l}_f\) from \(\bar{I}(B_{cd})\), and assume that \(\bar{l}_f\) does not overlap with its successor. Then, \(\bar{l}_f = l_f\), where \(l_f\) is the word in the \(f\)-th element of \(\bar{I}(B_{cd})\). Processor \(p_d\) uses invariants 4 and 5 to compute \(|\bar{l}_f|\) and the offset \(off_d\) of position \(d\) from the starting position of \(l_f\). Combined with the information stored in the head of cell \(B_{od+1}\) (cf. Invariant 3), this offset yields the position \(d'\) having offset \(off_d\) from the starting position of \(ls(B_{od+1})\). Thus, \(p_d\) knows that it is assigned to compare \(w_d\) with \(w_d\). If \(p_d\) detects a mismatch, it turns off a switch assigned to the starting position of \(l_f\). The case of an overlapping \(\bar{l}_f\) is handled similarly by the processors borrowed from \(\bar{l}_{f-1}\). At the end, at most one stub representative will have a switch still in the "on" position. If this is the case, such representative will identify itself by writing in the cell head. This concludes the description of the combination of left lists, which takes clearly constant time with \(\Theta(|B|)\) processors. Propagation of the invariants to \(B\) is trivial. □

Reasoning symmetrically, it is easy to introduce right stubs and right lists and so on in every cell partition of \(w\). This leads to establish a dual of Theorem 6. We use the term lexicographic lists refer to the collection of left and right lists. Theorem 6 and its dual admit the following corollary.

**Corollary 1** For any string \(w\) and integer \(\ell \leq |w|\), a CRCW PRAM with \(|w|\) processors can compute the lexicographic lists relative to the first \(\log \ell\) stages of the preprocessing of \(w\) in \(O(\log \ell)\) time and using linear auxiliary space per stage.

**Proof:** Straightforward. □

### 7.3 Standard representations for constant-time substring queries

A square-free string \(w\) together with the first \((\ell \leq |\log |w|| - 2)\) lexicographic lists is said to be in \(\ell\)-standard form. When \(\ell = |\log |w|| - 2\), we simply say that \(w\) is in standard
form. We are now ready to show that searching for a string in standard form into another string also in standard form is done instantaneously with a linear number of processors. With $y$ denoting a square-free pattern and $x$ a square-free text, we revisit the informal discussion at the beginning of Section 7.

Clearly, retrieving the seed $(i, s)$ of $y$ from its $|s|$-standard form is immediate. In fact, consider the partition of $y$ into cells of size $|s|$ and let $C$ be the cell of this partition which contains $i$.

**Fact 4** Stub $(i, s)$ is the first element of $\mathcal{H}(C)$.

Fact 4 is the handle to identify the position $i$ of $s$ in $y$. Since there are at most 4 cells in the partition of the positions of $y$, and each such cell contributes one known candidate, mutual comparison of the substrings of length $|s|$ starting at these candidate positions is all is needed. This is easily done in constant time with $|y|$ processors. There are of course more direct ways of computing the seed of $y$ within these bounds, but reasoning uniformly in terms of standard forms has other advantages in our context.

Assume to fix the ideas that $y$ is left-seeded, and that there is an occurrence of $y$ beginning in a cell $B$ of the partition of $x$ into cells of size $|s|$. Let $B'$ be the cell of $x$ where the corresponding occurrence of the seed $s$ begins (cf. Fig. 4). The identification of the position $j$ of $s$ within $B'$ is quite similar to the combination of adjacent left lists discussed earlier. In fact, $j$ is clearly the position of a left stub in $L(B')$. Lemma 9 of the previous section tells us that, if we consider the sequence, say, $\tilde{L}(B')$, then we can find at most one term $\tilde{l}_f$ such that $\tilde{l}_f$ is a prefix of $s$. We thus search in $\tilde{L}(B')$ for a term $\tilde{l}_f$ such that, letting $\tilde{s}$ be the prefix of $s$ of length $|\tilde{l}_f|$, we have that $\tilde{s} = \tilde{l}_f$. Lemma 10 and the discussion of Theorem 6 tell us that $O(|B|)$ processors are enough to match, simultaneously, each $\tilde{l}_f$-term against a corresponding prefix $\tilde{s}$ of $s$. The details are easily inferred from the preceding discussion and can be omitted.

Let now $y'$ be a substring of $y$, and consider the $(\log |y'|) - 2$-th lexicographic list for $y$. Clearly, $y'$ is embedded in a set of at most 9 consecutive cells in the associated cell partition of $y$. The same holds for every occurrence of $y'$ in any substring $x'$ of $x$ such that $|x'| \geq |y'|$. Again, assume to fix the ideas that $y'$ is left-seeded. Note that if $y'$ and its seed $(i', s')$ start in the same cell, say, $C'$ on $y$, it is no longer necessarily true that $(i', s')$ is the first term in the head list of $C'$. However, $(i', s')$ must still be a left stub in $C'$. Since the starting position $f$ of $y'$ in $y$ is known, all we need to do is to identify the leftmost left stub in $L(C')$ that starts at $f$ or to the right of $f$. This takes constant time with the priority-write emulation discussed in the introduction, after which we have a suitable substitute for Fact 4. From this point on, the search for $y'$ into $x'$ involves only minor variations with respect to the above description, and so does the search for $y'$ in any set of substrings of a given set of strings.
8 Bibliographic notes

Suffix trees are a special kind of the PATRICIA trees introduced by Morrison [1968]. The serial suffix tree construction presented in this paper is due to McCreight [1976]. An earlier construction, due to Weiner [1973], builds a variant of the tree known as position tree (cf. Aho, Hopcroft and Ullmann [1974]). Weiner's construction gave, as a trivial by-product, a linear-time method for finding the longest repeated substring in a string over a finite alphabet. Not long before, D. Knuth had posed the problem of whether such a problem could be solved in better than $O(n^2)$ time. Weiner's and McCreight constructions are equivalent at the outset, but they have notable intrinsic differences. Weiner's construction scans the input string from right to left, but does not need to know all of it before it can start. Conversely, McCreight's construction scans $x$ from left to right, but it needs the entire string before starting. The duality inherent to these two constructions was exposed by Chen and Seiferas [1985].

Subsequent constructions approach the problem of building the tree on-line (Majster and Majer [1985], Ukkonen [1992]), and build several variants such as inverted textfiles (Blumer et al., [1987]), factor transducers (Crochemore [1985], Blumer et al. [1985]), suffix arrays (Manber and Myers [1990]), etc.

The parallel construction of suffix trees presented in this paper is adapted from Apostolico et al. [1988]. It is an open problem whether this construction can be carried out in linear space an/or with $n/\log n$ processors when the alphabet is finite. The treatment of exhaustive on-line searches follows Apostolico [1992].

Suffix trees and their companion structures have found applications in many areas, including approximate string searching, data compression, computations of substring statistics and detection of squares and other regularities in strings. Some such applications are discussed in Apostolico [1985], and elsewhere in Apostolico and Galil [1985].

9 References


Crochemore, M. [1985]. "Optimal factor transducers", pp. 31-43 in [AG].


