JOURNAL OF COMPLEXITY **15**, 30–71 (1999)
Article ID jcom.1998.0496, available online at http://www.idealibrary.com on IDEAL®

Parallel Construction and Query of Index Data Structures for Pattern Matching on Square Matrices*

Raffaele Giancarlo†

Dipartimento di Matematica, Università di Palermo, Palermo, Italy

and

Roberto Grossi‡

Dipartimento di Sistemi e Informatica, Università di Firenze, Italy

Received June 26, 1997

We describe fast parallel algorithms for building index data structures that can be used to gather various statistics on square matrices. The main data structure is the Lsuffix tree, which is a generalization of the classical suffix tree for strings. Given an $n \times n$ text matrix A, we build our data structures in $O(\log n)$ time with n^2 processors on a CRCW PRAM, so that we can quickly process A in parallel as follows: (i) report some statistical information about A, e.g., find the largest repeated square submatrices that appear at least twice in A or determine, for each position in A, the smallest submatrix that occurs only there; (ii) given, on-line, an $m \times m$ pattern matrix PAT, check whether it occurs in A. We refer to the above two kinds of operations as queries and point out that they have applications to visual databases and two-dimensional data compression. Query (i) takes $O(\log n)$ time with $n^2/\log n$ processors and query (ii) takes $O(\log m)$ time with $m^2/\log m$ processors. The query algorithms are work optimal while the construction algorithm is work optimal only for arbitrary and large alphabets.

* An extended abstract related to this work was presented at the 5th Annual ACM Symposium on Parallel Algorithms and Architectures. Work supported in part by the Italian Ministry for Scientific Research and the Italian National Research Council.

† Part of this work was done while the author was a member of the technical staff at AT&T Bell Labs, U.S.A. E-mail: raffaele@altair.math.unipa.it.

‡ Part of this work was done while the author was visiting AT&T Bell Labs, U.S.A. E-mail: grossi@dsi.unifi.it.

0885-064X/99 $30.00

1. INTRODUCTION

In its simplest version, *two-dimensional pattern matching* extends classical string matching [41] as follows: given an $m_1 \times m_2$ *pattern* matrix PAT and an $n_1 \times n_2$ *text* matrix A with $m_1 \leqslant n_1$ and $m_2 \leqslant n_2$, one wants to find whether PAT occurs as a submatrix of A. It has applications in visual databases [34]. Another aspect of two-dimensional pattern matching is the gathering of statistical information about the text A, for example: Which one is the largest text submatrix appearing at least twice in A? For each position in A, what is the smallest text submatrix that occurs only there? These statistics have applications to two-dimensional data compression [42, 50, 52, 53]. We refer to this aspect as *two-dimensional statistics*. Let $M = m_1 \cdot m_2$ and $N = n_1 \cdot n_2$ be the number of elements in PAT and A, respectively. All the matrices in this paper have entries defined over a totally ordered alphabet Σ. We point out that most pattern matching algorithms are *alphabet dependent*, i.e., their running time depends both on the alphabet and input size, while others are *alphabet independent*, i.e., their running time depends only on the input size. Although the design of alphabet independent pattern matching algorithms is an interesting area of research as the difference between those two types of algorithms is important, for conciseness, we will mention results available in the literature without explicitly pointing out whether the algorithms are alphabet dependent or not.

The algorithms solving the two-dimensional pattern matching problem are roughly in two complementary classes: The ones that do not provide any statistical information about the text, and the ones that do. We briefly survey the main results for PRAM algorithms that fall in both classes. We refer the reader to [37] for the definition of the PRAM models of computation, in which the work is defined as the number of processors times the number of parallel time steps needed to complete the computational task at hand.

Pattern Matching Algorithms—No Statistics about the Text. They process pattern PAT first and then search for its occurrences in text A. An efficient CRCW PRAM algorithm in this class was first given by Mathies [44]. It uses $O(N)$ processors and $O(\log^2 N)$ time and there is no pattern processing. (Throughout the paper $\log x$ will indicate $\max(1, \log_2 x)$.) Amir and Landau [4] reduced the time to $O(\log N)$. Further improvements are due to Kedem *et al.* [38]. Their pattern processing takes $O(\log M)$ time with $O(M/\log M)$ processors and their search step takes $O(\log M)$ time with $O(N/\log M)$ processors. Amir *et al.* [1] devised a CREW algorithm in which the pattern processing can be done in $O(\log M)$ time with $O(M)$ processors and the search can be done in $O(\log N)$ time with $O(N/\log N)$

processors. For the CRCW PRAM model, Cole *et al.* [13] obtained an optimal pattern processing in $O(\log \log M)$ time with $O(M)$ work and an optimal search step in $O(1)$ time with $O(N)$ work. Finally, for the EREW PRAM, it has been shown by Czumaj *et al.* [18] that the optimal pattern processing takes $O(\log M)$ time with $O(M)$ work and the optimal search step takes $O(\log M)$ time with $O(N)$ work. For the more general case in which there is a set of patterns (called a dictionary) to be processed, and in which searching consists of finding all the occurrences of patterns of the dictionary in the text matrix, the reader can refer to [19, 21] for state of the art and references.

Pattern Matching Algorithms—Statistics about the Text. These algorithms build some auxiliary data structures for text A to gather some information about it. Then one can ask queries about the text. For instance one can ask, on-line, whether a given pattern PAT appears in the text A. Or, one can ask how many times a given submatrix of the text appears in it. There is only one set of PRAM algorithms in this class, due to Crochemore and Rytter [16], where the text and the pattern are constrained to be square matrices. Building their data structures for A requires $O(\log N)$ time with N processors in the CRCW PRAM. After that, one can look, on-line, for an occurrence of PAT in A in $O(\log M)$ time with $M/\log M$ processors. However, the most serious constraint is placed on the pattern size as it must be the product of powers of two. That is, PAT must have size $m_1 \times m_2$ with $m_1 = m_2 = 2^g$, for some integer $g \geqslant 0$.

1.1. *Our Results and Techniques*

The main contribution of this paper is to provide the first set of parallel algorithms for pattern matching with statistics about the text without the constraint on the pattern size. Our model of computation is the Arbitrary CRCW PRAM in which a set of synchronous processors have access in constant time to a common shared memory. In case of multiple writes to the same memory cell only one arbitrarily chosen processor succeeds [37]. We give some efficient algorithms for the construction and query of index data structures on the text, the main one being the *Lsuffix tree* of a square matrix [23]. This data structure represents all the square submatrices of a square matrix and it can be seen as a generalization of the suffix tree of a string [46].

The problem of building an index (tree) data structure for compactly storing *all* the submatrices of text A into its nodes, such that equal submatrices correspond to a common path from the root to an internal node, has been shown to be computationally harder than the one of building such a data structure for *only* the square submatrices of A [24]. Namely, the data structure for all the submatrices requires $\Omega(N^2/(\max(n_1, n_2)))$

space and therefore at least that amount of work is needed to build it. One such data structure has been considered in [25, 27], where a parallel construction requiring $O(\log N)$ time and $N^2/(\max(n_1, n_2))$ processors on an Arbitrary CRCW PRAM is given. Essentially, many suffix trees are built in parallel, each of which storing the submatrices having an identical number of rows (if $n_1 \leqslant n_2$) or columns (if $n_1 > n_2$). As for the square submatrices, we can do better than that and use only $O(N)$ space. From now on, we restrict attention to square matrices and square submatrices of a given matrix. In particular, we now have $n_1 = n_2 = n$ and $m_1 = m_2 = m$ (and therefore $N = n^2$ and $M = m^2$), and say that A and PAT have *side* n and m, respectively.

Construction of the Index Data Structures and Techniques Needed. We can build our data structures for text A in $O(\log N)$ time with N processors and $O(N^{1+\varepsilon})$ non-initialized space for any fixed constant ε, $0 < \varepsilon \leqslant 1$. In particular, our construction of the Lsuffix tree for A has optimal work when the alphabet Σ is arbitrary and large. (We wish to point out that the best known sequential solution for the construction of the Lsuffix tree requires $O(N \log N)$ time [23, 28] and that it is an open problem to find an $O(N)$ time construction for a small alphabet.) Our techniques are a non-trivial generalization of the classical ones devised by Apostolico *et al.* [8] for the parallel construction of the suffix tree of a string [46]. The main substantial difference between their algorithm and ours is the following. Essential to their algorithm is a labeling of substrings such that equal substrings get equal integer labels. As discussed in Subsection 3.4, some of their techniques used to compute those labels do not generalize to matrices because some very simple properties that hold for strings do not hold for matrices. In order to solve those problems, we use a new technique, which we refer to as *encapsulation* and that can be sketched as follows. When we need to assign a label to a given *chunk*, i.e., a piece of a matrix, we encapsulate it into a carefully selected submatrix of A, called a *capsular matrix*. Although two equal chunks do *not* necessarily have equal capsular matrices, we prove that this is so in our construction. The labels of the chunks are therefore computed on demand as a function of the capsular matrices.

For completeness, we point out that there are other known parallel algorithms in the literature to build a suffix tree of an m-length string: the randomized construction by Farach and Muthukrishnan [20] and the deterministic construction by Hariharan [32] and Sahinalp and Vishkin [49]. While Algorithm AILSV requires $O(m \log m)$ work and $O(\log m)$ time for any string alphabet, these others take $O(m)$ work and polylogarithmic time for a constant sized alphabet. It is an open problem to establish whether those algorithms extend to matrices.

Query Procedures and Techniques Needed. We denote by $A[i_1 : i_2, j_1 : j_2]$ the submatrix of size $(i_2 - i_1 + 1) \times (j_2 - j_1 + 1)$ identified by the entries in A at rows $i_1 \cdots i_2$ and columns $j_1 \cdots j_2$. So $A = A[1 : n, 1 : n]$. We adopt the convention that $A[i, j] = \$$ for $i > n$ or $j > n$, where $\$ \notin \Sigma$ is a special character not appearing elsewhere. Once we have built our data structures for text A, we can find quickly some information about the "structure" of A with suitable queries. Indeed, we can solve the following problems in $O(\log N)$ time with $N/\log N$ processors:

- Build a compacted weighted vocabulary by storing implicitly how many times each submatrix appears in A. The queries take a pair (i, j) and an integer k as input, and output the number of other pairs (r, s) such that $A[i : i + k - 1, j : j + k - 1] = A[r : r + k - 1, s : s + k - 1]$.
- Find the largest repeated submatrices. Namely, find the largest k for which there are two distinct pairs (i, j) and (r, s) such that $A[i : i + k - 1, j : j + k - 1]$ and $A[r : r + k - 1, s : s + k - 1]$ are equal. Output k and one such pair.
- Compute the submatrix identifiers. For each pair (i, j), find the smallest k such that $A[i : i + k - 1, j : j + k - 1] \neq A[r : r + k - 1, s : s + k - 1]$ for any other pair (r, s).
- Determine the submatrix that appears most frequently among the ones having side at least h, for a given integer $h > 0$.

We need some more insight into the combinatorial structure of our index data structures to answer, on-line, the following pattern matching query in $O(\log M)$ time with $M/\log M$ processors:

- Given pattern PAT, check whether or not PAT occurs as a submatrix of A. This amounts to finding whether there is an integer pair (i, j) such that $A[i : i + m - 1, j : j + m - 1]$ and PAT are equal entry by entry.

We show in this paper that the time and processor bounds for these queries on our index data structures give an optimal work bound. For the pattern matching query, the bound depends only on the pattern size M, i.e., the number of elements in it. We point out that obtaining an optimal bound for the pattern matching query is not straightforward and consists of avoiding the repeated examination of the same pieces of A while looking for PAT. This is sufficient to notify that we have found a pattern occurrence. In order to list all the pattern occurrences, we must report all the integer pairs (i, j) such that $A[i : i + m - 1, j : j + m - 1]$ and PAT are equal. As we shall see, the latter task reduces to a standard tree computation in our case and we therefore omit its discussion.

1.2. *Relation of the Lsuffix Tree with the Suffix Tree for Strings*

As previously mentioned, the Lsuffix tree extends to higher dimension the notion of a suffix tree for strings [46]. As the latter is a central data structure in many applications involving strings, it is quite appealing to see whether an analogous data structure can be used for problems involving matrices. The answer is positive for many problems, although their applicability is limited as of now. In order to illustrate this point, let us draw an analogy between the developments that have involved the suffix tree for strings and the state of the art for the Lsuffix tree.

When the suffix tree for strings appeared in the literature, the few problems in which it was used were motivated either by being "interesting" from the combinatorial point of view or by data compression. This point is well illustrated in [6, 41]. For instance, Knuth, Morris, and Pratt [41] posed the following combinatorial problem: find a linear time algorithm that identifies the longest repeated substring of a string. They also conjectured that no such algorithm exists. Weiner [55] solved the posed problem via a new data structure: The Position Tree (a variant of the suffix tree). It is worth mentioning that the work by Weiner was motivated by a data compression problem. Although the versatility of the suffix tree was clear to the specialists [5], only recently a larger community of researchers (especially the ones interested in computational biology) has became aware of it. Good sources for the state of the art on suffix trees and related applications are [7, 17, 30].

The motivation for the introduction of the Lsuffix tree closely follows the one for the suffix tree. It can be used to solve some problems for matrices that are interesting from the combinatorial point of view and some others that are definitely relevant in applications. In what follows, we state further problems that can be solved with the use of the Lsuffix tree and then address their relevance to applications.

- Compute statistics for data compression. For any two pairs (i, j) and (r, s), given on-line, find the largest submatrix $A[r : r+k-1,\ s : s+k-1]$ that appears in $A[1 : i, 1 : j]$ in constant work. (This is the basic search step in LZ-type compression algorithms [53].)

- Find the smallest k-repeat. That is, find the submatrix that appears exactly k times for a given integer k. (The problem for strings is described in [30].)

- Determine the longest common (prefix) submatrix. For any two pairs (i, j) and (r, s), given on-line, find their largest common submatrix in constant work, i.e., compute the largest side k such that $A[i : i+k-1,\ j : j+k-1] = A[r : r+k-1, s : s+k-1]$ (or output $k = 0$ otherwise).

- Build the suffix array by producing an ordered list of "suffix positions" belonging to A. (It requires an additional vector storing the longest common prefix of some suffixes in order to search for patterns quickly. See [39, 43] for practical algorithms building the suffix array.)

- Search in multiple texts. Namely, given matrices $A_1, A_2, ..., A_t$, build some index data structures for them so that PAT can be searched for to detect which matrices A_j contain PAT, for $1 \leqslant j \leqslant t$. (This problem has been considered on strings in [45].)

- Search in a dictionary. Given patterns $PAT_1, PAT_2, ..., PAT_p$, build some dictionary data structures for them so that a text matrix A can be queried to know which patterns PAT_i occur in A, for $1 \leqslant i \leqslant p$. (It is symmetrical to the previous problem and there are specific techniques for its solution [19, 21].)

- Determine the longest common submatrix of two or more matrices. Given matrices $A_1, A_2, ..., A_t$, compute the side $s(k)$ of the largest submatrix common to at least k of the matrices, for $2 \leqslant k \leqslant t$. (This problem has been solved for strings in [33].)

- Find all-against-all occurrences. Given matrices $A_1, A_2, ..., A_t$ find whether A_j occurs in one of $A_1, A_2, ..., A_{j-1}, A_{j+1}, ..., A_t$, for $1 \leqslant j \leqslant t$.

Although the above problems may be "interesting" from the combinatorial point of view, it is unclear whether or not they will all have a wide range of applications as their counterparts on strings. For example, determining the longest common submatrix for two pairs (i, j) and (r, s) extends to higher dimension the problem of finding the longest common prefix of two suffixes in strings. While the latter problem is a fundamental tool for many problems in exact and approximate string matching [30], to the best of our knowledge, we cannot say the same for exact and approximate multidimensional pattern matching.

Nevertheless, there are some areas in which our data structures might be helpful. In visual databases for multimedia systems, 2-D strings [11] are employed to retrieve images by their contained objects according to their relative position. Each image is partitioned into quadrants, so that the resulting representation is a square matrix in which each quadrant corresponds to a matrix entry. If the object is in quadrant (i, j), a proper token representing the object is put in matrix entry (i, j). All the remaining (empty) entries are filled with a special "empty" token. The 2-D strings are a possible representation of the matrix by reading the nonempty matrix entries in a suitable order with the aim of building some index data structures. There are three possible levels for querying these data structures. The index data structures in this paper are particularly useful for the so-called level 2, which can be equivalently stated in terms of our pattern matching

query. That is, we look for the images having certain objects in given positions of a smaller region while preserving their relative order and position in the image. This corresponds to an exact pattern matching search for an input matrix that we are able to treat very efficiently. The other two levels, level 0 and level 1, require an approximate search that we do not know how to perform quickly with our indexing techniques.

Finally, in data compression [53], the text is compressed by means of backward references to the already examined part of the text. Finding the largest submatrix starting from the current text position, such that the submatrix appears in the already examined part, is the most time consuming task and our index data structures provide efficient support to queries of this kind.

1.3. *Organization*

The remainder of this paper is organized as follows. In Section 2 we describe some basic tools that assist us in the task of assigning labels to specific submatrices of A. We refer to this process as *naming*. We then present our index data structures in Section 3 and our on-line pattern matching algorithm in Section 4. The last section contains some concluding remarks and open problems.

2. BULLETIN BOARDS AND NAMING

Our model of computation is the Arbitrary CRCW PRAM in which a set of k synchronous processors $P_1, ..., P_k$ have access to a common shared memory in constant time. In case of multiple writes to the same memory cell only a single (arbitrarily chosen) processor succeeds [37] and we refer to it as *the winner*. Throughout the computation described in this paper, each processor P_i knows its index i and can access a common table of size $k \times k$, called the *Bulletin Board*, which is not initialized. The access to a Bulletin Board is ruled as follows. When different processors attempt to write in the same location of the Bulletin Board, only the winner succeeds. Apostolico *et al.* [8], Crochemore and Rytter [16], and Sahinalp and Vishkin [49] used Bulletin Boards to perform parallel computations on strings and matrices. We abstract two computational problems, *Tuple Labeling* and *Matrix Naming*, whose solutions make use of Bulletin Boards and are needed later on in our algorithms. Tuple Labeling deals with assigning equal integer labels to equal tuples and is mainly needed by Matrix Naming, where we assign equal integer labels to equal submatrices. We point out that the algorithms we present here are based on very well known pattern matching techniques. Therefore, we limit ourselves to sketch the main points of the algorithms.

2.1. *Tuple Labeling*

The Problem. Given tuples $t_1, ..., t_k$, each tuple consisting of $s = O(1)$ integer components in the range $[1 \cdots k]$, we want to assign equal integer labels to equal tuples in parallel (the assigned labels are in $[1 \cdots k]$).

Outline of the Solution. We use processors $P_1, ..., P_k$, where tuple t_i is held by P_i, for $1 \leqslant i \leqslant k$. Each processor P_i is in charge of assigning an integer label in the range $[1 \cdots k]$ to its tuple t_i, such that $t_i = t_j$ if and only if they have equal labels. Each P_i performs its task inductively in $s-1$ stages. Except for the first stage, the output of a given stage is the input for the next stage. In each stage, P_i assigns a temporary label to t_i and then reduces the number of components in t_i by one. At the end, t_i contains a single component, which is the final label for the original tuple t_i. All stages are the same, therefore we describe only the base stage.

Processor P_i takes the last two tuple components a, b in t_i and attempts to write its index i into entry (a, b) of a Bulletin Board. As all the processors work synchronously on the same Bulletin Board, only the winner succeeds in writing its index, say j, in entry (a, b). Subsequently, each P_i reads j from that entry of the Bulletin Board and replaces the last two components a and b by the temporary label j in its own tuple t_i. As a result, a, b are always replaced by an identical (temporary) label j in all tuples whose last two components are a, b and, by induction, equal tuples are still equal while unequal tuples are still unequal. At this point, processors start another inductive stage on the reduced tuples, each having $s-1$ components.

Remark 2.1. After the last stage, each tuple is reduced to a single integer and, by induction, only equal tuples get an identical integer. Since we do not initialize Bulletin Boards, it is worth noting that "garbage entries" are implicitly created: they are the unused entries in the Bulletin Board in which no processor writes/reads. Moreover, the Tuple Labeling algorithm also works with tuples having integer components in $[1 \cdots \text{poly}(k)]$ by splitting each component into $O(1)$ smaller components of $\lfloor \log_2 k \rfloor$ bits each or less, so that the total number of components is still $O(1)$.

Time and Space Analysis. It is easy to show that the entire computation takes $O(s) = O(1)$ time with k processors. The space requirement is $O(k^2)$. However, there are several standard ways to reduce the $O(k^2)$ work space required by each Bulletin Board. For instance, one can use the technique devised by Apostolico *et al.* [8] to reduce the space to $O((1/\varepsilon)\, k^{1+\varepsilon})$, for $0 < \varepsilon \leqslant 1$, by increasing the time complexity by a constant factor $O(1/\varepsilon)$. One can also apply the algorithm by Bhatt *et al.* [10] for sorting $O(k)$ integers in $[1 \cdots \text{poly}(k)]$ and prefix sums [15] to simulate each synchronous access to a Bulletin Board. The time slow-down factor is $O(\log k / \log \log k)$,

but the space reduces to $O(k)$. Finally, Amir *et al.* [3] have introduced a class of hash functions that allow one to reduce the space to $O(k)$ with an $O(\log^* k)$ time slow-down factor and no significant increase in the work bound. The use of hash functions makes the algorithm randomized. In this paper, we adopt the technique by Apostolico *et al.* [8].

A Partial Function for Tuples. We point out that Tuple Labeling implicitly defines a partial function from tuples with s entries to integers in the range $[1 \cdots k]$: Let us assume that we have processed tuples $t_1, ..., t_k$ in the $s-1$ stages of Tuple Labeling and we have kept $s-1$ distinct Bulletin Boards, one for each stage. The implicit partial function is defined for a new tuple t with s entries as follows. If t equals one of the processed tuples $t_1, ..., t_k$, say $t = t_j$, we assign t_j's label to t. Otherwise, t is different from the others and we return a failure.

Computing the Partial Function for a New Tuple. Once that Tuple Labeling has been carried out, the computation of the partial function for a new tuple can be done in $O(s) = O(1)$ time with a single processor by reading the proper entries in the Bulletin Boards previously set for this purpose. The only relevant implementation detail is to check for the "garbage" entries in each such Bulletin Board (recall that we do not initialize the Bulletin Boards). If we find garbage, t is different from the other tuples and so we return a failure; otherwise, we go on in the computation for t. We use a standard trick to check for garbage, namely we maintain a cross-reference to the entries that are actually written during the computation for the processed tuples [31].

2.2. *Matrix Naming*

The Problem. Given an $n \times n$ matrix A whose entries are characters taken from the alphabet $\Sigma = \{1, ..., O(n^2)\}$, we want to assign equal integer labels, called *names*, to equal submatrices of A having a power-of-two side: $A[i : i + 2^r - 1, j : j + 2^r - 1]$ for $1 \leqslant i, j \leqslant n$ and $0 \leqslant r \leqslant \lceil \log n \rceil$. (Names are integers in $[1 \cdots n^2]$ stored into a vector of size $O(n^2 \log n)$.) We use the previously mentioned convention that $A[f, g] = \$$ for $f > n$ or $g > n$, where \$ is a special character not appearing elsewhere (e.g., we can encode \$ by integer 0).

Outline of the Solution. Crochemore and Rytter [16] have devised a parallel algorithm for the Matrix Naming problem by implementing the Karp–Miller–Rosenberg sequential technique [36]. The algorithm proceeds "inductively" in successive stage as follows.

At the base stage $r = 0$, we apply the Tuple Labeling algorithm (with $k = n^2$), in which the tuples are the single entries $A[i, j]$.

At the inductive stage $r > 0$, we assume that stage $r-1$ has been executed. A matrix having side 2^r is equally partitioned into four non-overlapping matrices having side 2^{r-1}. Their respective names a, b, c, d are known by stage $r-1$ and form a corresponding quadruple (a, b, c, d), so that any two matrices having side 2^r are equal if and only if their corresponding quadruples are identical. For this reason we run the Tuple Labeling algorithm on these quadruples and the resulting integer labels become the names of the matrices having side 2^r. When $r = \lceil \log n \rceil$, the task is completed.

Time Analysis and Extensions. Each stage of the algorithm takes $O(1)$ time with n^2 processors. Therefore, we can obtain the required names for matrices in $O(\log n)$ time with n^2 processors. A well-known variant of this approach can be used to assign names to arbitrary matrices. Given any matrix M having side ℓ, let 2^r be the largest power of two smaller than ℓ and divide M into its four *covering submatrices* having side 2^r: they appear at M's corners and their union with the overlaps gives M. We then form a quintuple given by ℓ and the four names known by the Matrix Naming algorithm. We execute the Tuple Labeling algorithm on these quintuples and let $name(M)$ be the resulting integer assigned to M. We have that equal matrices get an identical name. This further step requires constant time when the names for the covering submatrices having side 2^r are available.

A Partial Function for Matrices. Analogously to the Tuple Labeling algorithm, a partial function from matrices to their names is implicitly defined here. Let us assume to have computed and stored (in a vector of $O(n^2 \log n)$ size) the names of the submatrices of A produced by the Matrix Naming algorithm, and let us assume also that we have kept all the Bulletin Boards used in this task. The implicit partial function is defined for a new matrix Q as follows. If Q equals one of the processed submatrices, say $Q = M$, we assign M's name to Q, i.e., $name(Q) = name(M)$. If Q is different from all the submatrices, we let $name(Q)$ be a special value that is always different from the other names including itself. We say that the name of Q is *consistent* if it satisfies the above properties.

Computing the Partial Function for a New Matrix. Given a new matrix Q, the task essentially consists of computing its consistent name by reading the proper entries in the Bulletin Boards previously set for this purpose and by checking for the "garbage" entries. The cost of this computation is either $O(|Q|)$ work and $O(\log |Q|)$ time from scratch [16], or $O(1)$ work if the names of the four *covering* submatrices of Q are known (see above for the definition of covering submatrices).

3. INDEX DATA STRUCTURES FOR THE TEXT

Here we introduce the Lsuffix tree and some additional data structures related to it. The presentation is organized as follows. We start with some preliminary observations about the suffix tree of a string [46] that may be of help in understanding the definition of Lsuffix tree. We then define this data structure in Subsection 3.2. In Subsection 3.3, we present three basic problems in which the Lsuffix tree can be used and the corresponding algorithms. The next three subsections are devoted to the presentation of the algorithm for the construction of the suffix tree. Finally, the last subsection presents additional data structures that will be used in conjunction with the Lsuffix tree for an efficient implementation of the pattern matching query. Presentation of the algorithm for this latter task is given in Section 4.

3.1. *Preliminaries*

A standard data structure used in many applications of string algorithms is the suffix tree [46]. Good sources for the state of the art on the suffix tree and its applications are [7, 17, 30]. Informally speaking, a suffix tree for a string x is a compacted trie or digital search tree [40, 47] that stores suffixes $x[i:|x|]$, $1 \leqslant i \leqslant |x|$, into its leaves as shown in Fig. 1. Let us assume without any loss of generality that the last character in x is a special symbol \$ not appearing elsewhere. The suffix tree for x satisfies the following conditions: Each arc from parent to child is labeled by a substring of x and, for each suffix $x[i:|x|]$, there is a path from the root to a leaf whose concatenation of labels gives the suffix at hand. Since all the suffixes of x are distinct, there is a one-to-one correspondence between the leaves of the suffix tree and the suffixes of x. When storing a suffix tree, any substring $x[i:j]$ labeling an arc is actually represented by integer pair (i, j) as shown in Fig. 1. Many applications of suffix trees for various problems on strings are listed in [6, 7, 17, 30]. For example, since any given string y occurring in x has a corresponding path in the suffix tree for x (e.g., string "ba" in the suffix tree in Fig. 1), we can report all the occurrences of y in x by listing positions in x corresponding to the leaves descending from the aforementioned path (e.g., leaves "2" and "4" in Fig. 1).

We would like to follow a similar approach when dealing with matrices. For this reason, we use a "linear representation" of matrices to store them into a suitable compacted trie, the *Lsuffix tree* [23]. In the next subsection we introduce the required notions.

3.2. *The Lsuffix Tree*

Lstrings. We now describe *Lstrings*, which are linear representations of matrices introduced in [2, 23]. We begin by giving an informal discussion

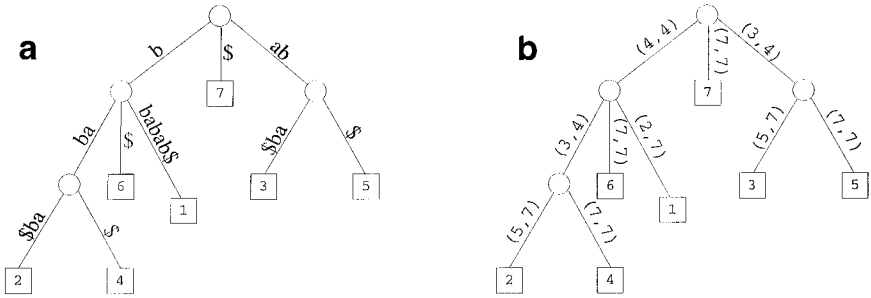

FIG. 1. The suffix tree for string $x = bbabab\$$ and its compact representation. The characters labeling the arcs in (a) should be read top-down. Leaf i corresponds to suffix $x[i: |x|]$, for $1 \leqslant i \leqslant 7$.

for a matrix $A[1:n, 1:n]$ with reference to Fig. 2a. We divide A into n L-shaped pieces, the ith one being composed of subrow $A[i, 1:i-1] = A[i, 1]\ A[i, 2] \cdots A[i, i-1]$ and subcolumn $A[1:i, i] = A[1, i]\ A[2, i] \cdots A[i, i]$. We concatenate $A[i, 1:i-1]$ and $A[1:i, i]$ together and take the resulting sequence of $2i-1$ characters into consideration as being an atomic and indivisible string called the ith *L-character*. As a result, we can represent A by listing its first L-character $A[1, 1]$, its second L-character $A[2, 1]\ A[1, 2]\ A[2, 2]$, its third L-character $A[3, 1]\ A[3, 2]\ A[1, 3]\ A[2, 3]\ A[3, 3]$, and so on, as shown in Fig. 2b. That is, we get a sequence of Lcharacters which is called *Lstring*.

More formally, let $L_\Sigma = \bigcup_{i=1}^{\infty} \Sigma^{2i-1}$ be the *alphabet of Lcharacters*, each of which is an atomic and indivisible string over Σ. We can treat Lcharacters suitably by exploiting their definition in terms of strings. Two Lcharacters are

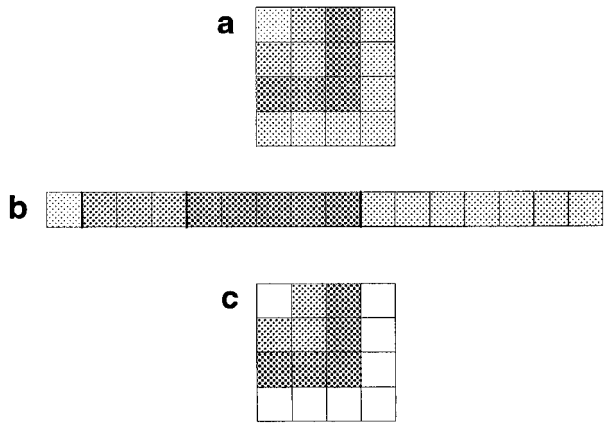

FIG. 2. (a) A matrix $A = A[1:4, 1:4]$ divided into Lcharacters (= same shading); (b) its linear representation as Lstring; (c) a chunk composed of the 2nd and 3rd Lcharacters.

equal if and only if they are equal as strings over Σ. Two Lcharacters can be concatenated under the following restriction: an Lcharacter in Σ^{2i-1} can *precede* only an Lcharacter in Σ^{2i+1} and *succeed* only one in Σ^{2i-3}. Any number of Lcharacters whose concatenation satisfies the above restriction is a *chunk* (see Fig. 2c). An *Lstring* $\alpha[1:n]$ is a chunk such that the first Lcharacter is in Σ. We denote the chunk composed of the Lcharacters at positions $h \cdots j$ in α by $\alpha[h:j]$. The length of chunk $\alpha[h:j]$ is the number $j-h+1$ of Lcharacters composing it. We denote the length of a chunk β by $|\beta|$. Two chunks $\alpha[h:j]$ and $\beta[h:j]$ are equal if and only if $\alpha[k]=\beta[k]$, for $h \leqslant k \leqslant j$. (The previous definitions clearly apply to Lstrings too.) The definition of Lstrings is most easily understood in terms of their natural correspondence to square matrices. As it should be clear from Fig. 2, Lstrings are intended to represent matrices while chunks are intended to represent L-shaped pieces of matrices centered around the main diagonal.

Tries for Lstrings. We now wish to store some suitable Lstrings into a trie over the alphabet L_Σ so as to obtain the Lsuffix tree. We need the notion of trie built on Lstrings and illustrate it by means of an example. Let us consider three matrices X, Y, and Z in Fig. 3a and take their corresponding Lstrings. A trie for these Lstrings is shown in Fig. 3b. It is like a trie for strings except for the following characteristics: Each arc is labeled by an Lcharacter; all the arcs departing from a node are labeled by different Lcharacters, all belonging to Σ^{2i-1} where i is the number of traversed nodes from the root; there is a leaf v for each Lstring at hand, such that the concatenation of the Lcharacters along the path from the root to v gives the Lstring; if two Lstrings have their first k Lcharacters identical then they share a common path of k arcs from the root. Tries for Lstrings can have one-child nodes just like tries for strings. We therefore obtain a compacted trie for Lstrings by compressing the longest paths of one-child nodes into single arcs. Their new labels are chunks obtained by concatenating the Lcharacters found along the compressed paths. The resulting data structure is a compacted trie for Lstrings as shown in Fig. 3c, where sibling arcs are labeled by chunks whose first Lcharacters are distinct.

The Definition of the Lsuffix Tree. We can now describe the Lsuffix tree for matrix A by carefully choosing its "suffixes," i.e., which submatrices should have their Lstrings stored in a compacted trie. Recall that we adopt the convention that $A[f, g]=\$$ for $f>n$ or $g>n$, where \$ is a special character not appearing elsewhere and not matching itself, i.e., each instance of \$ is different from the others. According to this convention, we use A_{ij} to denote the $n \times n$ submatrix in A whose top leftmost entry is (i, j) (i.e., $A_{ij}=A[i:i+n-1, j:j+n-1]$) and let the "suffixes" of A be all such matrices A_{ij}, with $1 \leqslant i, j \leqslant n$. These matrices are distinct due to the \$'s and we use α_{ij} to denote the Lstring corresponding to a given A_{ij} (we

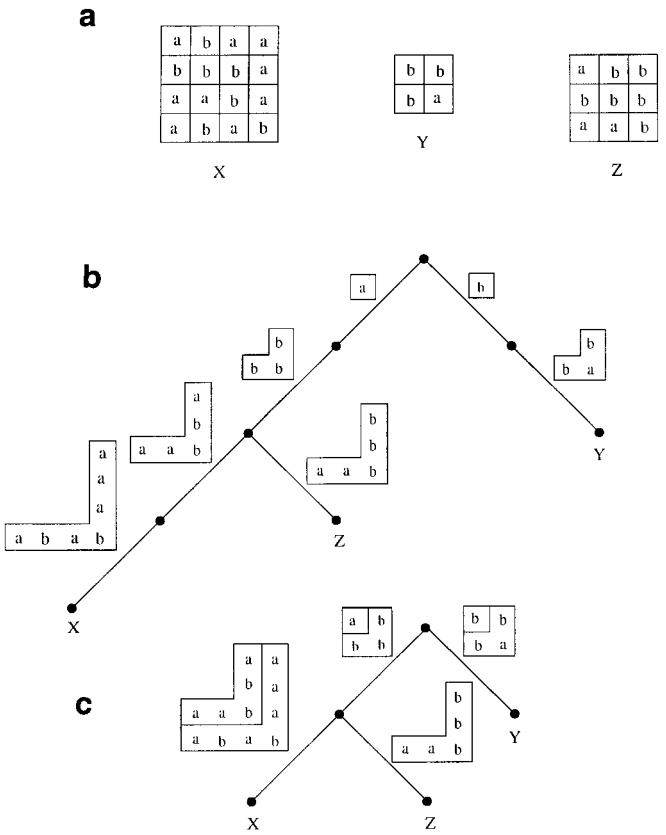

FIG. 3. (a) Three matrices X, Y, and Z, and (b) the trie built on their Lstrings; (c) the compacted version of the trie.

implicitly assume that $1 \leqslant i, j \leqslant n$). The Lsuffix tree is then the compacted trie built on Lstrings α_{ij}, in which the leaf storing a given α_{ij} is identified by label (i, j). For example, a part of the Lsuffix tree for the matrix in Fig. 4 (top) is shown in Fig. 4 (middle), where only some of the \$'s are shown. In analogy with the suffix tree for a string, we can represent a chunk $\alpha_{ij}[p : q]$ labeling a suffix tree arc by means of a *descriptor*, i.e., an integer quadruple (i, j, p, q) computable in constant time (see Fig. 5). We obtain the compact representation of the Lsuffix tree shown in Fig. 4 (bottom). We refer the reader to [23] for more details, where it is shown that the Lsuffix tree has $O(n^2)$ nodes and takes optimal $O(n^2)$ space in memory.

We use the following terminology for the Lsuffix tree. We say that a node u is the *locus* of an Lstring β if and only if the label concatenation

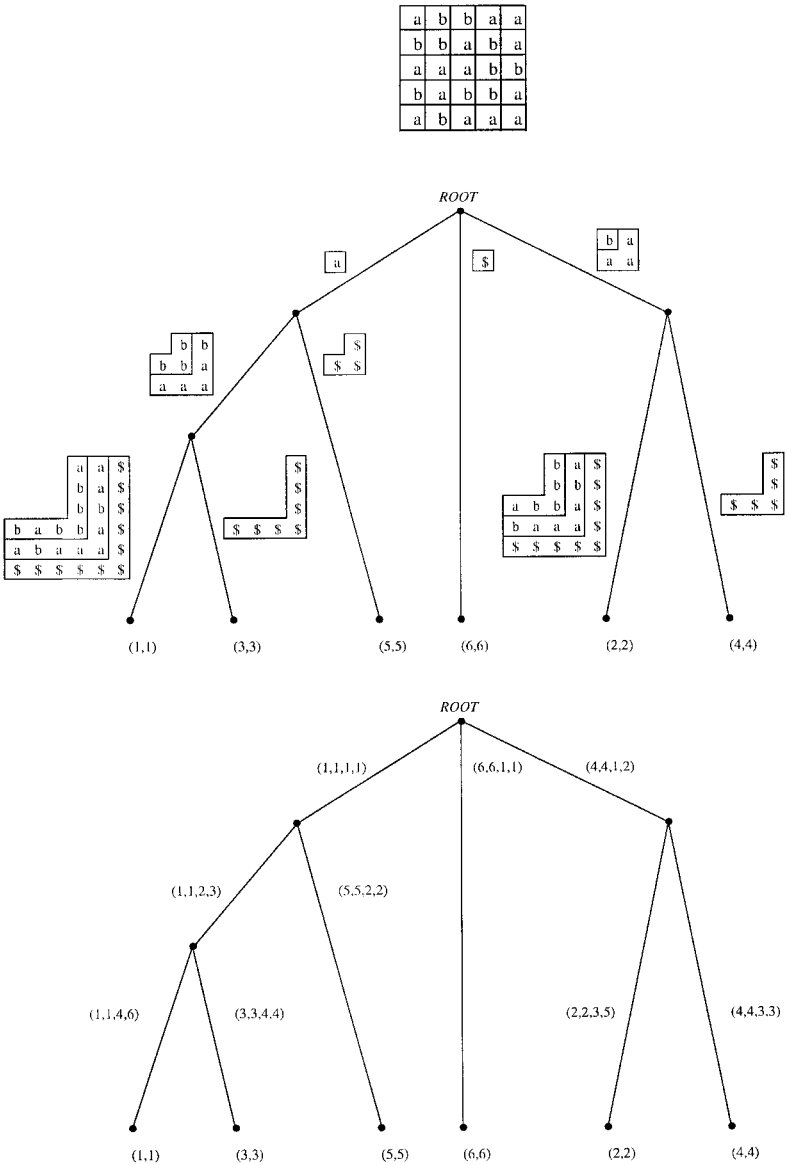

FIG. 4. Top, a matrix A. Middle, the part of the Lsuffix tree for the Lstrings α_{ii} with $i = 1, ..., n$, corresponding to the main diagonal of A. Bottom, its compact representation with descriptors.

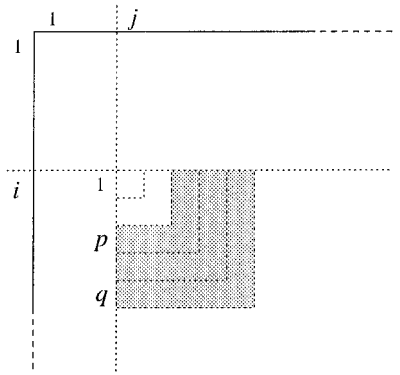

FIG. 5. Chunk $\alpha_{ij}[p:q]$ represented by descriptor (i, j, p, q).

along the downward path leading from the root to u is equal to β. An extension of β is an Lstring whose first $|\beta|$ Lcharacters are equal to β. The *extended locus* of β is the locus of the shortest extension of β having its locus in the Lsuffix tree. If this extended locus exists, then it is unique.

LEMMA 3.1 [23]. *Let T_A be the Lsuffix tree for a matrix A and β be the Lstring corresponding to any given matrix B. Matrix B is equal to one of the submatrices in A if and only if the extended locus of β is in T_A. All the positions in A in which B appears are given by the labels in the leaves descending from this extended locus.*

3.3. *Some Uses of the Lsuffix Tree and the Corresponding Algorithms*

Before going on to describe how to build the Lsuffix tree, we show how to apply it to collect statistical information about the submatrices of a text matrix A of size $n \times n$. Let us assume we have already processed A and built its Lsuffix tree T_A. Then we can take the following problems into consideration:

(P1) *Compact weighted vocabulary.* For each submatrix of A provide the number of its occurrences by means of an implicit index representation of A. The queries take a pair (i, j) and an integer k as input, and output the number of other pairs (r, s) such that $A[i:i+k-1, j:j+k-1] = A[r:r+k-1, s:s+k-1]$.

(P2) *Largest repeated submatrices.* Find the largest submatrices that occur at least twice in A. That is, find the largest k for which there are two distinct pairs (i, j) and (r, s) such that $A[i:i+k-1, j:j+k-1]$ and $A[r:r+k-1, s:s+k-1]$ are equal. Output k and one such pair. Variations of this problem involve, for example, finding the submatrices of A that occur most frequently and are of side at least h, for some integer $h > 0$.

(P3) *Submatrix identifier.* For each position (i, j) of matrix A, find the smallest submatrix that occurs only there. That is, find the smallest k such that $A[i : i+k-1, j : j+k-1] \neq A[r : r+k-1, s : s+k-1]$ for any other position (r, s).

A few remarks are in order. We do not discuss the remaining applications mentioned in the Introduction as many of them can be solved analogously to Problems (P1)–(P3) and within the same work bounds. Crochemore and Rytter [16] describe some parallel algorithms that provide a solution to Problem (P2) by a different processing of A, and the time-processor bounds of their algorithms are comparable with those presented in this paper.

In order to solve (P1)–(P3), we need the following information for each node u in T_A: the number $\chi(u)$ of leaves in the subtree rooted at u; the length $\lambda(u)$ of the Lstring whose locus is u (i.e., $\lambda(u)$ is the sum of the lengths of the chunks labeling the nodes along the path from the root to u). It is now quite straightforward to solve problems (P1)–(P3) by using T_A and the information just mentioned.

(P1) *Solution.* Let u be the extended locus in T_A of the Lstring α corresponding to $A[i : i+k-1, j : j+k-1]$, where $|\alpha| = k$. This extended locus can be identified by taking the leaf with label (i, j) in T_A and its shallowest ancestor u such that $\lambda(u) \geqslant k$. By Lemma 3.1, the leaves descending from u represent all the occurrences of A' in A. Hence, their number is $\chi(u)$.

(P2) *Solution.* We use the fact that a submatrix A' occurs at least twice in A if and only if the extended locus in T_A of the corresponding Lstring α' is not a leaf. In order to produce the largest repeated submatrices we therefore take the internal nodes u in T_A such that $\lambda(u)$ is maximum. Let $\bar{\lambda}$ be this maximum value and u be any node such that $\lambda(u) = \bar{\lambda}$. A largest repeated submatrix is then the $\bar{\lambda} \times \bar{\lambda}$ submatrix that corresponds to the Lstring having locus in u. We therefore output $k = \bar{\lambda}$ and the pair (i, j) labeling a leaf descending from u. All this information is available in the descriptor labeling the arc that links u to its parent as it must be a quadruple of the form $(i, j, p, \bar{\lambda})$ for some non-negative integer $p \leqslant \bar{\lambda}$. By using the same techniques, we can answer queries such as finding the most frequently repeated submatrices of side greater than or equal to an integer $h > 0$. We take the subset of internal nodes u (if any) such that $\lambda(u) \geqslant h$, and then select the ones for which $\chi(u)$ is maximum.

(P3) *Solution.* We use the fact that there is a one-to-one correspondence between the positions (i, j) of A and the leaves of T_A. Given a leaf l, let (i, j) be its label and u be its parent node. The submatrix identifier in position (i, j) is then given by the submatrix $A[i : i+k-1, j : j+k-1]$

where $k = \lambda(u) + 1$. In other words, it is the smallest submatrix among those occurring only in position (i, j).

THEOREM 3.2. *Given an $n \times n$ matrix and its Lsuffix tree, Problems* (P1)–(P3) *can be solved in $O(\log n)$ time with $n^2/\log n$ processors on an Arbitrary CRCW PRAM. Moreover, any on-line query in Problem* (P1) *can be answered in $O(\log n)$ time by a single processor.*

Proof. The correctness follows from Lemma 3.1. For the computation of integers $\chi(u)$ and $\lambda(u)$, we adopt standard parallel techniques [35, 37] such as Euler tour [54], prefix sums [22], and list ranking [14]. The Euler tour of a tree T is the circuit obtained by replacing each edge of T by two arcs of opposite direction and by traversing all such directed arcs so that no arc is traversed twice. The prefix sums of n elements $x_1, x_2, ..., x_n$ under a binary associative operation $\odot$ are given by $x_1 \odot x_2 \odot \cdots \odot x_i$, for every $1 \leqslant i \leqslant n$. The list ranking problem is to determine the distance from each node in a list from the end of the list. As far as the time and processor bounds are concerned, computing $\chi(u)$ and $\lambda(u)$ for each node u in a tree with $N = n^2$ nodes takes $O(\log N)$ time with $N/\log N$ processors. Moreover, the maximum in a set of $O(N)$ elements (i.e., the integers stored in the nodes of T_A) can be found in $O(\log \log N)$ time with $N/\log \log N$ processors (e.g., see [51]). It remains to see how to perform a query in Problem (P1). This amounts to detecting u, the extended locus in T_A of the Lstring corresponding to $A[i : i + k - 1, j : j + k - 1]$. We can find u by a binary search on the ancestors of leaf labeled (i, j), until we find the shallowest ancestor u such that $\lambda(u) \geqslant k$. As the hth ancestor can be found in constant time after a linear work processing of T_A [9], the cost is $O(\log n)$ time with a single processor. ∎

3.4. *Parallel Construction of the Lsuffix Tree: Outline*

We now discuss how to build the Lsuffix tree in parallel. The Lstrings representing the matrix "suffixes" A_{ij} allow us to re-use some of the ideas presented for strings and suffix trees in the algorithm by Apostolico, Iliopoulos, Landau, Schieber, and Vishkin [8] (shortly, AILSV). In this section we show how to adapt AILSV for our purposes. In the next subsection we describe the modifications required to make it work on Lstrings without any loss in efficiency.

The key notion in Algorithm AILSV for an m-length string is that of *refinement*: it produces a sequence of $O(\log m)$ trees denoted $D^{(r)}$ (for $r = \lceil \log m \rceil, ..., 0$), each one of which is a better approximation of the suffix tree and takes $O(m)$ work and constant time to be built. We show that this notion of refinement can be smoothly applied to the matrices thanks to their Lstring representation. However, a chunk comparison during the

refinement is an expensive operation that we have to avoid in order to get efficiency. We only give the details of the most significant steps in our algorithms. A more technical and detailed discussion is in [26].

We are given an $n \times n$ matrix A on which we want to build the Lsuffix tree T_A. We assume without any loss of generality that n is a power of two and that the matrix entries are characters taken from the universe $\{1, ..., O(n^2)\}$ (if not, they can be sorted and consistently numbered in $O(\log n)$ time with n^2 processors [12]).

Given any two chunks $\alpha[p:q]$ and $\beta[p':q']$, we say that they have *refiner* ℓ if their first ℓ Lcharacters are equal (i.e., $p = p'$ and $\alpha[k] = \beta[k]$, for $p \leqslant k \leqslant p + \ell - 1$). We now define the intermediate trees that allow us to build the Lsuffix tree. A *refinement tree* $D^{(r)}$ for the suffix tree is a labeled tree satisfying the following constraints (for $0 \leqslant r \leqslant \log n$):

(D1) There are n^2 leaves labeled by the pairs (i, j), $1 \leqslant i, j \leqslant n$, and no internal node having one child, except for the root. Each leaf (i, j) corresponds to the Lstring α_{ij} representing "suffix" A_{ij}.

(D2) Each node is labeled by a chunk (at least 2^r long) represented by a descriptor, so that the concatenation of the labels along the downward path from the root to any leaf (i, j) equals its corresponding Lstring α_{ij}. If descriptor (i, j, p, q) labels a node, then leaf (i, j) is its descendant (we recall that (i, j, p, q) describes chunk $\alpha_{ij}[p:q]$ as shown in Fig. 5).

(D3) Any two chunks labeling sibling nodes start with Lcharacters of identical size and, furthermore, do not have refiner 2^r. (Constraint (D2) makes it possible to apply the definition of refiner $\ell = 2^r$.)

We wish to point out that the arcs in $D^{(r)}$ are stored as child-to-parent pointers and its *nodes* are labeled by chunks, whereas Lsuffix tree *arcs* are parent-to-child pointers labeled by chunks. At the beginning, $D^{(\log n)}$ is made-up only of the root and the n^2 leaves, with leaf (i, j) labeled by descriptor $(i, j, 1, n)$; at the end, $D^{(0)}$ satisfies the definition of Lsuffix tree except that the labels must be moved from the nodes to the arcs and the direction of the arcs must be inverted. We give an example of a sequence of refinement trees in Figs. 6–8. At the beginning, $D^{(\log n)}$ can be built in $O(n^2)$ work and constant time. The next important task is to transform $D^{(r)}$ into $D^{(r-1)}$ by means of the following procedure. We let the children of a node in $D^{(r)}$ be referred to as its *nest*. We say that two nodes $u, v \in D^{(r)}$ are *equivalent* if and only if u and v are in the same nest and the chunks labeling them have refiner 2^{r-1}.

PROCEDURE TRANSFORM (r). It produces $D^{(r-1)}$ from $D^{(r)}$ by the following two steps.

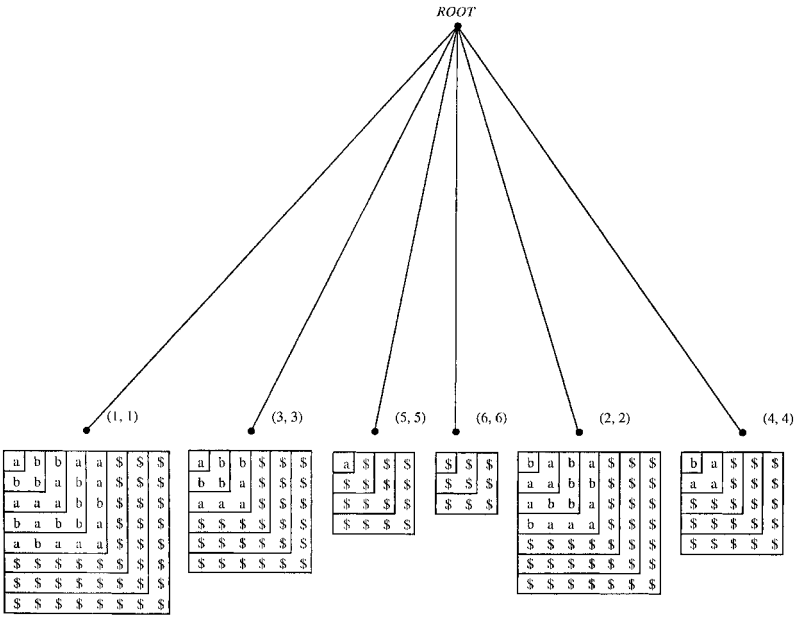

FIG. 6. The refinement tree $D^{(2)}$ for the Lstrings α_{ii}, $i = 1, ..., 6$, in matrix A shown in Fig. 4(top). Here n is not a power of two ($n = 5$), and some \$'s are not shown.

Step 1. We partition the nodes of $D^{(r)}$ into equivalence classes (according to the equivalence relation just defined). For each equivalence class $\mathscr{C}$ with $|\mathscr{C}| > 1$, we create a new node w. The parent u of the nodes in $\mathscr{C}$ becomes the parent of w, and w becomes the new parent of the nodes in $\mathscr{C}$. If (i, j, p, q) is the descriptor labeling a node in $\mathscr{C}$, then we assign label $(i, j, p, p + 2^{r-1} - 1)$ to w and change the third component of the descriptor from p to $p + 2^{r-1}$, for each node in $\mathscr{C}$.

Step 2. Let $\tilde{D}^{(r)}$ be the tree resulting from Step 1. For each node u (other than the root) whose nest produced only one equivalence class, we remove u from $\tilde{D}^{(r)}$ and make the only child w of u be a child of the parent of u. We modify their labels as follows: If (i, j, p, q) and $(i', j', q + 1, q')$ are the descriptors labeling u and w, respectively, then the descriptor of w becomes (i', j', p, q'). The resulting tree is $D^{(r-1)}$.

LEMMA 3.3. *Procedure Transform*(r) *correctly transforms* $D^{(r)}$ *into* $D^{(r-1)}$.

Proof. The only non-trivial part of the proof consists of showing that the removal of one-child nodes in Step 2 does not apply simultaneously to a node u and its parent v in $\tilde{D}^{(r)}$. This guarantees that if u is removed from $\tilde{D}^{(r)}$ because it has a single child w, its parent v still exists in that tree and

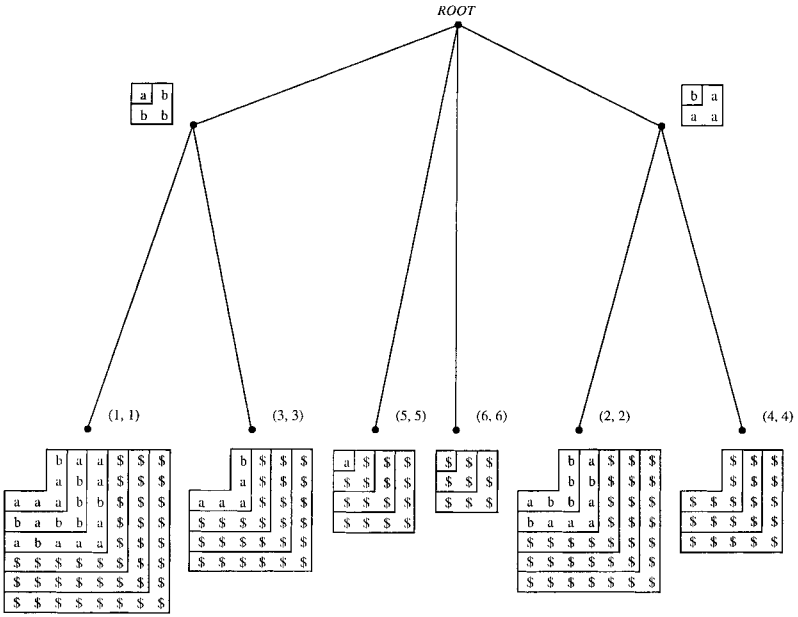

FIG. 7. The refinement tree $D^{(1)}$ obtained from the one in Fig. 6.

can be made the new parent of w. In order to prove that v cannot be removed during Step 2 we show that v has always at least two children by considering two cases:

Case 1. v is also the parent of u in $D^{(r)}$. We know that v cannot be a new node generated during Step 1 and it has a nest with at least two nodes in $D^{(r)}$, by definition of $D^{(r)}$. Moreover u belongs to a singleton equivalence class $\mathscr{C}$, i.e., $|\mathscr{C}| = 1$. Consequently, the nest of v in $D^{(r)}$ has been divided into at least two equivalence classes, implying that it has at least two children.

Case 2. v differs from the parent of u in $D^{(r)}$. That is, v is a new node generated at the end of Step 1 to become the new parent of the nodes in a class $\mathscr{C}$ with $|\mathscr{C}| > 1$. Therefore v has at least two children. ▮

We now prove a property satisfied by the nodes (except the root) of $D^{(r)}$ that will be useful for the design of the on-line pattern matching query.

LEMMA 3.4. *Given a refinement tree $D^{(r)}$, the length of the chunk labeling any node other than the root is either 2^r or at least 2^{r+1}.*

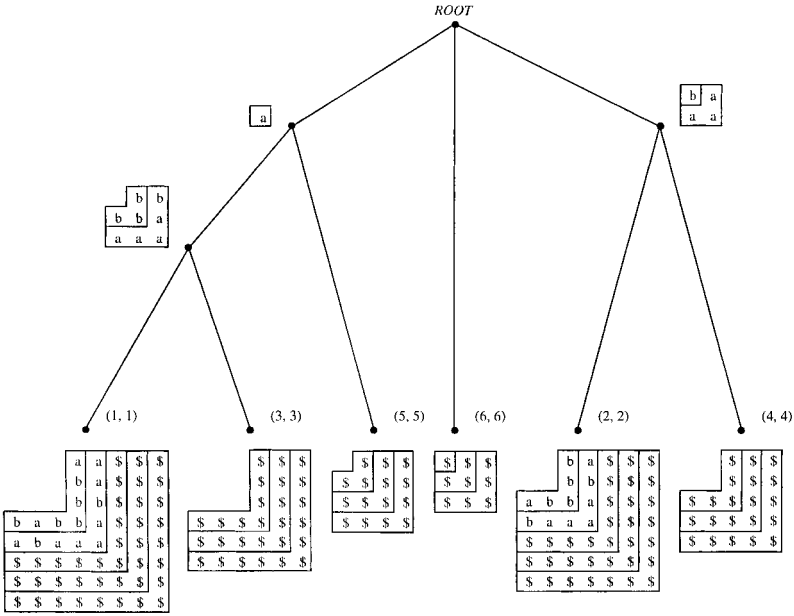

FIG. 8. The final refinement tree $D^{(0)}$ obtained from the one in Fig. 7. Note that $D^{(0)}$ is equivalent to the Lsuffix tree shown in Fig. 4(middle).

Proof. The lemma holds by induction. It is true for $r = \log n$ as $D^{(\log n)}$ has only a root and n^2 leaves storing Lstrings of length n. Let us therefore assume it holds for $r+1$ and show it holds for r. We first examine the intermediate tree $\tilde{D}^{(r+1)}$ obtained after the execution of Step 1. We wish to point out that each node u in it, except the root, is labeled by a chunk of length either equal to 2^r or at least 2^{r+1}. In order to see why, we discuss two possible cases. If u is created in $\tilde{D}^{(r+1)}$ during Step 1, its chunk has length 2^r by the way the label is assigned to it. Otherwise, the length of the chunk is either unchanged or gets reduced by 2^r: by induction, it was previously either 2^{r+1} or at least 2^{r+2} and so it becomes either 2^r or at least 2^{r+1}. Next, let us examine how $D^{(r)}$ is obtained from $\tilde{D}^{(r+1)}$ in Step 2. The length of a chunk either remains unchanged or increases by at least 2^r, and we therefore satisfy the inductive property claimed: each internal node in $D^{(r)}$ is labeled with a chunk of length either 2^r or of length at least 2^{r+1}. ∎

3.5. *Parallel Construction of the Lsuffix Tree: Some Details*

The parallel implementation of Steps 1 and 2 in Procedure Transform(r) follows the one adopted in AILSV to build the refinement trees. We refer

the reader to [8] to see how to do that in linear work and constant time when dealing with strings. In particular, the processor allocation policy in [8] allows us to describe the algorithms as if each node had a unique processor allocated to it. However, what makes our computation difficult is that the refinement steps must work on Lstrings rather than strings. This causes no problem for Step 2. Indeed, it is a simple exercise to show that the techniques presented in [8] can be extended to implement Step 2 in $O(n^2)$ work and constant time. The problem we encounter in Step 1 is how to partition the nodes of $D^{(r)}$ into equivalence classes in $O(n^2)$ work and constant time. We first outline some simple solutions to this problem that are based on known techniques. Unfortunately, they do not satisfy the needed work bound. Then, we outline our technique. Therefore, the remaining part of this subsection deals with the implementation of Step 1.

There are $\Theta(n^2)$ nests and each nest seems to require $O(n^2)$ work to be partitioned with a brute force approach, while the rest of the operations in Steps 1–2 can be performed in $O(n^2)$ work and constant time according to [8]. The total is $O(n^4)$ work.

We could reduce the partitioning work by using the Karp–Miller–Rosenberg pattern matching technique [16, 36] adapted to Lstrings. We briefly describe this approach. First, a name is assigned to the chunks (of a power-of-two length) that can be found in the Lstrings, so that any two chunks are equal if and only if they both have the same name. Each node in $D^{(r)}$ also receives a unique integer identifying it. It is worth noting that the names and these integers are in the range from 1 to $O(n^2)$. Second, a pair (η_1, η_2) is assigned to each node u (except the root) where η_1 is the unique integer assigned to the parent of u and η_2 is the name assigned to the first 2^{r-1} Lcharacters in the chunk labeling u. The aim of η_2 is to have a constant time check for refiner 2^{r-1}. As a result, the equivalent nodes have equal pairs and the node partitioning can be obtained in $O(n^2)$ work and constant time by executing the Tuple Labeling partition algorithm described in Subsection 2.1. The drawback to this approach is that it requires us to process and assign names to $\Omega(n^3 \log n)$ *distinct* chunks of a power-of-two length in the worst case (when matrix A is composed of $\Theta(n^2)$ distinct characters). The total work is reduced to $O(n^3 \log n)$.

It is worth noting that not all the chunks of a power-of-two length need the names because the ones that need them vary with the refinement step used. Since we do not know in advance which chunks are actually needed, we have to precompute names for all of them. As a result, the direct computation of their names is too expensive. In contrast, the direct computation of the names for the $O(n^2 \log n)$ submatrices of a power-of-two side can be efficiently done with the Matrix Naming algorithm described in Subsection 2.2. However, we cannot use these names in a standard way. For example, we cannot take a chunk and split its matrix entries into "maximal"

submatrices whose names are known. Since the chunks may represent also matrix parts with small aspect ratios (e.g., long and thin), a chunk may have a tuple of $O(n)$ names associated in the worst case with $O(n^3 \log n)$ total work (there may be $O(n^2)$ nests in each of the $O(\log n)$ refinement steps, and each nest may contain some tuples of $O(n)$ names).

Our Technique. Fortunately, we introduce a technique that can be used to partition the nodes in $O(n^2)$ work. It is based on the notion of a *capsular matrix* for a chunk, which is the smallest submatrix that encloses the matrix part represented by the chunk. We present the technique by first defining the capsular matrices and stating a few facts about them and then by describing how to use capsular matrices.

Capsular Matrices. Given a chunk with descriptor (i, j, p, q), we define its capsular matrix as the $q \times q$ submatrix appearing in the top leftmost corner of "suffix" A_{ij} (i.e., submatrix $A[i : i+q-1, j : j+q-1]$). Equivalently, the Lstring representing the capsular matrix is given by the first q Lcharacters of the Lstring α_{ij} representing A_{ij}.

Use of Capsular Matrices. We use the capsular matrices to check for refiner 2^{r-1}. Specifically, let δ_u denote the first 2^{r-1} Lcharacters in the chunk labeling a node $u \in D^{(r)}$ other than the root (it is well defined by condition (D2) of the definition of refinement trees) and assume that $(i, j, p, p+2^{r-1}-1)$ is its descriptor for some integers i, j, p. The definition of δ_u is motivated by the fact that the chunks labeling two nodes u and v in the same nest have refiner 2^{r-1} if and only if $\delta_u = \delta_v$. Let us examine the capsular matrix M_u for δ_u. It is the $(p+2^{r-1}-1) \times (p+2^{r-1}-1)$ submatrix appearing in the top leftmost corner of "suffix" A_{ij} and so its Lstring is given by the first $(p+2^{r-1}-1)$ Lcharacters of α_{ij}. Fact 3.5 and Lemma 3.6, stated next, allow us to compare chunks δ_u and δ_v for any two nodes u and v by means of their capsular matrices M_u and M_v (in general, this is not possible for two arbitrary chunks).

FACT 3.5. *Given any node $u \in D^{(r)}$ other than the root, the Lstring representing the capsular matrix M_u has extended locus in u.*

Proof. As a result of our assumption on the descriptor of chunk δ_u, the chunk labeling u is represented by descriptor (i, j, p, q) for a suitable positive integer $q \geq p+2^{r-1}-1$. By condition (D2) of the definition of $D^{(r)}$, there is a leaf descending from node u that is the locus of Lstring α_{ij}. Let σ be the Lstring representing M_u. Since σ is given by the first $(p+2^{r-1}-1)$ Lcharacters of α_{ij}, we have that the former is a prefix of the latter. Let us examine the Lstring γ having locus in u: It is equal to the first q Lcharacters of α_{ij}. Having $q \geq p+2^{r-1}-1$ immediately implies that σ is a prefix of γ. We now examine the Lstring λ having locus in *parent*(u): The

fact that λ is shorter than σ gives that λ is a proper prefix of σ and so γ is the shortest extension of σ having a locus (in u). Consequently, u is the extended locus of σ. In the special case $q = p + 2^{r-1} - 1$, the two Lstrings σ and γ are equal and u is their locus. ▌

LEMMA 3.6 (Encapsulation). *For any two distinct nodes u and v in $D^{(r)}$, not equal to the root, we have that $M_u = M_v$ if and only if u and v are equivalent (i.e., u and v are in the same nest and the chunks labeling them have refiner 2^{r-1}).*

Proof. According to our assumption on chunks δ_u and δ_v, let $(i_u, j_u, p_u, p_u + 2^{r-1} - 1)$ and $(i_v, j_v, p_v, p_v + 2^{r-1} - 1)$ be their descriptors, respectively. We remark that $\delta_u = \delta_v$ if and only if the chunks labeling u and v have refiner 2^{r-1}.

($\Rightarrow$) Let us assume that $M_u = M_v$. They both have sides of equal length and so $p_u = p_v = p$. This, and the fact that their Lstrings are equal, give that the Lcharacters in positions $p, ..., p + 2^{r-1} - 1$ are pairwise identical. That is, $\delta_u = \delta_v$. We now prove that nodes u and v are in the same nest. Let σ and τ be the (same length) Lstrings corresponding to M_u and M_v, respectively. By Fact 3.5, σ and τ have extended locus in u and v, respectively. Consequently, $\sigma = \sigma' \delta_u$ and $\tau = \tau' \delta_v$, where σ' and τ' are the (possibly empty) Lstrings whose locuses are $parent(u)$ and $parent(v)$, respectively. Since $M_u = M_v$, we have $\sigma = \tau$; the fact that $\delta_u = \delta_v$ implies $\sigma' = \tau'$. Now, the locus in $D^{(r)}$ of an Lstring is unique (because otherwise condition (D2) or (D3) of the definition of $D^{(r)}$ would be violated). This gives $parent(u) = parent(v)$ and so u and v are in the same nest.

($\Leftarrow$) Let us assume that u and v are equivalent, i.e., $\delta_u = \delta_v$ and $parent(u) = parent(v) = w$. Hence, we have that $p_u = p_v = p$ for a positive integer p, where p_u and p_v are in the descriptors of δ_u and δ_v. Let λ be the Lstring whose locus is w (its length is $p - 1$) and let Z be the matrix represented by $\lambda \delta_u = \lambda \delta_v$. Then $M_u = Z = M_v$ as their (equal length) Lstrings are equal. This follows from the fact that: (i) their first $p - 1$ Lcharacters are surely equal (to λ) because their paths all go through node w by Fact 3.5, and (ii) their remaining Lcharacters are equal as we know that $\delta_u = \delta_v$ by hypothesis. ▌

Lemma 3.6 can be used to find a partition of the nodes in $D^{(r)}$ into equivalence classes by just collecting their corresponding *equal capsular matrices* together. Its importance therefore is in the fact that it reduces an expensive task—finding a node partition under a certain equivalence relation—to a simpler task which can now be solved with a standard computation: the identification of equal matrices in a given set. We next describe how to perform that task *under the assumption* that the names of

square submatrices of A of side a power of two have been computed as described by the Matrix Naming algorithm in Subsection 2.2.

Identification of Equal Matrices in a Given Set. For each node $u \in D^{(r)}$ other than the root, we locate its corresponding chunk δ_u and therefore its capsular matrix M_u in constant time. Chunk δ_u is well defined as its length is 2^{r-1} and the chunk labeling u is surely of length at least 2^r by condition (D2) of the definition of refinement trees. We let $\mathscr{H}$ denote the set of $O(n^2)$ matrices thus obtained. Since we are assuming that we have computed the names of the submatrices in A having a power-of-two side, the names of the four covering submatrices of each matrix $Q \in \mathscr{H}$ must have been computed by the Matrix Naming algorithm in Subsection 2.2. We can therefore compute the partial function $name(Q)$ in constant work for each $Q \in \mathscr{H}$ and assign names to all the matrices in $\mathscr{H}$ in total $O(n^2)$ work and constant time. We refer the reader to Subsection 2.2 for the necessary definitions and details. As a result, we get a partition of the nodes because equivalent nodes have capsular matrices with the same name as a result of Lemma 3.6.

Finally, we point out, omitting the details, that we perform the rest of Step 1 in Procedure Transform(r) in $O(n^2)$ work and constant time. As already pointed out, Step 2 takes the same amount of work and time by using techniques analogous to the ones reported in [8].

LEMMA 3.7. *Procedure Transform(r) takes $O(n^2)$ work and $O(1)$ time to transform $D^{(r)}$ into $D^{(r-1)}$, assuming that we have computed the names of the submatrices of A having a power-of-two side.*

3.6. *Correctness and Time Analysis*

We now outline a proof of correctness and provide a time analysis of our parallel Lsuffix tree construction:

THEOREM 3.8. *Given an $n \times n$ matrix A, its Lsuffix tree can be built in $O(\log n)$ time with n^2 processors on an Arbitrary CRCW PRAM.*

Proof. The correctness can be proved by induction. The construction of tree $D^{(\log n)}$ is the base step. Lemma 3.3 gives the correctness for the inductive step from $D^{(r)}$ to $D^{(r-1)}$. As for the complexity, the parallel algorithm implementing the Karp–Miller–Rosenberg technique [36] can be applied to the submatrices of a power-of-two side in A in $O(\log n)$ time with n^2 processors, so that a submatrix name can be retrieved in constant time by a single processor (see Subsection 2.2). Tree $D^{(\log n)}$ can be built in constant time with n^2 processors and each tree $D^{(r-1)}$ can be obtained from $D^{(r)}$ in

constant time with n^2 processors by Lemma 3.7. The execution of $O(\log n)$ of these steps takes a total of $O(\log n)$ time with n^2 processors. We recall that the processor allocation policy is the one adopted in Algorithm AILSV. ▮

3.7. *Building the Index Data Structures for Text*

We now describe how to collect the information on the text matrix A for pattern matching purposes. The central data structure is the Lsuffix tree T_A for A. There are also three other data structures that must be kept together with the Lsuffix tree. Their use will be clear in Section 4. We anticipate that the first two auxiliary data structures can be obtained as a byproduct of Lsuffix tree construction. The third one requires some discussion which we give after its description.

(1) We maintain all the intermediate refinement trees $D^{(r)}$, $0 \leqslant r \leqslant \log n$, produced by the parallel Lsuffix tree construction described in Subsection 3.4. We revert the direction of the arcs and store them as parent-to-children pointers into a proper (not initialized) array, one array per node. We now describe which position is assigned in the array to each arc. Let us assume that $OUT(v)$ is the array for a node $v \in D^{(r)}$ and we have to represent an arc (v, u) linking v to one of its children u. We take the first 2^r Lcharacters of the chunk labeling u and locate its capsular matrix M'_u. It is worth noting that this matrix is defined as the capsular matrix M_u used in the Lsuffix tree construction, with the difference that M'_u is used for the first 2^r Lcharacters of the chunk labeling u while M_u is taken for the first 2^{r-1} Lcharacters of that chunk. We then compute each $name(M'_u)$ in constant work by the partial function described in Subsection 2.2 as the names of the four covering of M'_u are known. We represent arc (v, u) as a pointer to u and store it in position $name(M'_u)$ of $OUT(v)$. We wish to point out that no two arcs departing from v get the same position in $OUT(v)$ because this means that the two corresponding capsular matrices would have equal names and so the two chunks would have refiner 2^r (contradicting condition (D3) of the definition of refinement trees). Moreover, each name is an integer in the range $[1 \cdots n^2]$ and so the total size $\sum_v |OUT(v)|$ of these arrays for a single refinement tree does not exceed the one of a single Bulletin Board.

We also need to connect the nodes of the different refinement trees by means of *thread* links. Given a node $u \in D^{(r)}$, its thread link points to the copy of u in $D^{(r-1)}$. If this copy does not exist because u became a one-child node in going from $D^{(r)}$ to $D^{(r-1)}$, we mark $u \in D^{(r)}$ and its thread link points to the node $w \in D^{(r-1)}$ that was its only child in $\tilde{D}^{(r)}$ (see Steps 1 and 2 of Procedure Transform).

(2) We keep all the $O(\log n)$ Bulletin Boards needed to compute names (see Section 2). In particular, we need those used for: (i) assigning names to the submatrices of a power-of-two side that were processed by the Matrix Naming algorithm in Subsection 2.2 for the Lsuffix tree construction; (ii) computing $name(M'_u)$ for the capsular matrices M'_u needed in the refinement trees mentioned in point (1) above. As previously remarked in Subsection 2.2, keeping these Bulletin Boards allows us to compute a partial function to assign consistent names to new matrices.

(3) We store the names of some "small" text submatrices into two tries $\mathscr{R}$ and $\mathscr{C}$. Let us examine the submatrices in point (2) and take only those whose side is 2^s for $0 \leqslant s \leqslant \log L$ and a parameter $L = O(\sqrt{\log n})$ to be specified later on. We denote their set by S. Given a matrix B, let $row(B)$ and $col(B)$ be the strings obtained by reading the characters in B in row and column major order, respectively. Trie $\mathscr{R}$ stores strings $row(B)$ into its nodes, for all $B \in S$. Trie $\mathscr{C}$ is defined analogously on strings $col(B)$, for $B \in S$. We do not compress the tries, i.e., we allow them to have one-child nodes: every prefix of the stored strings has therefore a locus. We record each node f that is locus of a string $row(B)$ or $col(B)$. It may happen that f corresponds to several (equal) matrices B, but it is sufficient to record just one of those matrices for our purposes. We augment $\mathscr{R}$ and $\mathscr{C}$ with the following auxiliary information for each $B \in S$: (a) We store $name(B)$ into the locus of $row(B)$ and $col(B)$; (b) we set a *cross link* from the locus of $row(B)$ to the locus of $col(B)$ and vice versa; (c) we set a *shortcut link* from the locus of $row(B)$ to the locus of $row(B')$, where B' is B without its first row (an analogous link is set from $col(B)$ to $col(B'')$, where B'' is B without its first column). We remark that the shortcut links are well defined for the matrices in S.

Outline of the Algorithm for the Construction of $\mathscr{R}$ and $\mathscr{C}$. The algorithm for the construction of $\mathscr{R}$ and $\mathscr{C}$ with input parameter L has $\log L + 1$ phases and requires n^2 processors, one processor for each entry (i, j) in matrix A. At the beginning, $\mathscr{R}$ and $\mathscr{C}$ are only made up of their roots. Let us assume that during the first $s-1$ phases, the versions of $\mathscr{R}$ and $\mathscr{C}$ that represent only the matrices in S of side no more than 2^{s-1} have been built. We describe phase s.

During phase $s \leqslant \log L$, the processor in charge of entry (i, j) examines B_{ij}, the submatrix of side 2^s whose top leftmost entry is (i, j), and inserts in parallel $row(B_{ij})$ into $\mathscr{R}$ and $col(B_{ij})$ into $\mathscr{C}$, for all $1 \leqslant i, j \leqslant n$. We outline how the insertion is done into $\mathscr{R}$ and how the links are set up in $\mathscr{R}$ (the procedures are analogous for $\mathscr{C}$). We have 2^{2s} inductive steps in phase s that work as follows.

In the base step $q = 0$, all processors are at the root of $\mathscr{R}$ as it is the locus of the empty string. Let us assume inductively that each processor has correctly reached, in step $q-1$, the locus w of the first $q-1$ characters of $row(B_{ij})$ (this is trivially true for $q-1=0$). Let c be the qth character in $row(B_{ij})$. In step q, if there is a branch out of w labeled c, the processor synchronously follows it to move to the next node. Otherwise, we have to create this branch: the processor at hand competes with the other processors, if any, willing to branch from w using c. The winner creates the missing node and labels the corresponding arc with c. This new arc is then traversed by all competitors (see [38] for a detailed implementation of the operations for the parallel construction of tries). After the final step $q = 2^{2s}$, the current node w is the locus of $row(B_{ij})$. Another processor competition takes place to record w and store $name(B_{ij})$ into it. The winner also sets the cross and shortcut links for w by traversing again the tries.

Remark 3.9. Phase s of the construction of $\mathscr{R}$ and $\mathscr{C}$ takes $O(2^{2s})$ time with n^2 processors as each branch can be implemented in constant time [38]. Therefore, the total time for the construction of the data structures in point (3) is $O(\sum_{s=0}^{\log L} 2^{2s}) = O(L^2)$. As $L = O(\sqrt{\log n})$, we obtain $O(\log n)$ time with n^2 processors.

We can now state the total cost of building the data structures in points (1)–(3):

THEOREM 3.10. *Building the index data structures for an $n \times n$ text matrix A takes $O((1/\varepsilon)\log n)$ time and n^2 processors on an Arbitrary CRCW PRAM, with a total of $O((1/\varepsilon)\, n^{2+\varepsilon})$ non-initialized space, for any given constant $\varnothing < \varepsilon \leqslant 2$.*

Proof. The time and processor bounds for building the data structures in points (1) and (2) are those of the Lsuffix tree construction in Theorem 3.8, namely, $O(\log n)$ time with n^2 processors. By Remark 3.9, building the data structures in point (3) takes the same bounds. As far as the space complexity is concerned, this is dominated by the $O(\log n)$ Bulletin Boards, each of which has size $k \times k$ and so requires $O(k^2)$ space, where $k = O(n^2)$. We reduce this space as in Section 2. In particular, we can store each Bulletin Board in $O((1/v)\, k^{1+v})$ space, for any constant $0 < v \leqslant 1$, with a time slow-down of $O(1/v)$ in our algorithms. We obtain $O((1/v)\log n)$ time and a total space of $O(\log n(1/v)\, k^{1+v}) = O(\log n(1/v)\, n^{2+2v})$. For any given constant $0 < \varepsilon \leqslant 2$, we can fix $v = \varepsilon/4$, so that the required time becomes $O((1/\varepsilon)\log n)$ and the total space becomes $O(\log n(1/\varepsilon)\, n^{2+\varepsilon/2}) = O((1/\varepsilon)\, n^{2+\varepsilon})$. A similar approach can be taken to reduce the space required by the arrays OUT in the refinement trees. ∎

4. PATTERN QUERY

In this section we deal with the following two-dimensional on-line pattern matching problem: We are given an $n \times n$ text matrix A on which we can build some index data structures. We then take, on-line, an $m \times m$ pattern matrix PAT, $m \leqslant n$, whose entries are a subset of those in A and want to check to see if PAT equals a submatrix of A. A common variant of this problem asks for identifying all the submatrices equal to PAT. We solve our pattern query problem by searching for the extended locus of the Lstring π representing PAT in some of the refinement trees produced during the construction of the Lsuffix tree for A. This is sufficient to notify that we have found a pattern occurrence. In order to list all the pattern occurrences, we must report all the leaves descending from that locus. This is a standard tree computation and therefore we omit its presentation. Our pattern query is organized in two main parts and makes use of the index data structures in Subsection 3.7 (Theorem 3.10) as follows.

ALGORITHM PATTERN MATCHING. Let π be the Lstring representing PAT, and let us decompose π into "maximal" chunks $\pi_0, \pi_1, ..., \pi_k$, $k \leqslant \lfloor \log m \rfloor$, such that they all have a power-of-two length and their concatenation gives $\pi = \pi_0 \pi_1 \cdots \pi_k$. Namely, π_0 is composed of the first 2^{r_0} Lcharacters in π, where 2^{r_0} is the largest power of two smaller than or equal to m; π_1 is composed of the next 2^{r_1} Lcharacters in π, where 2^{r_1} is the largest power of two smaller than or equal to $m - 2^{r_0}$; we go on until we have $m - \sum_{i=0}^{k} 2^{r_i} = 0$. We now have two main parts, where we implicitly assume that $0 \leqslant i \leqslant k$:

Part I. Compute each $name(PAT_i)$, where PAT_i is the capsular matrix of chunk π_i. By definition of capsular matrix, we know that $PAT_i = PAT[1 : \ell_i, 1 : \ell_i]$, where $\ell_i = \sum_{j=0}^{i} 2^{r_j}$, and its Lstring is given by the first ℓ_i Lcharacters in π (see Fig. 9). We cannot apply directly the partial naming function described in Subsection 2.2 to assign consistent names as we would process $\Theta(m^2 \log m)$ submatrices in PAT whereas we want to spend only $O(m^2)$ total work. On the other hand, there are $O(\log m)$ capsular matrices PAT_i to process, each of which has $\Theta(m^2)$ size. In Subsection 4.1, we describe how to compute their names in a total of $O(\log m)$ time with $m^2/\log m$ processors.

Part II. Search in the refinement trees. The search is intuitively simple and proceeds by induction on $i = 0, ..., k$ by using the decomposition of $\pi = \pi_0 \pi_1 \cdots \pi_k$. We start out at the root of $D^{(r_0)}$ with $i = 0$ and find the locus of π_0. In the inductive step $i > 0$, we have found the locus of $\pi_0 \pi_1 \cdots \pi_{i-1}$ in $D^{(r_{i-1})}$. We reach the locus of $\pi_0 \pi_1 \cdots \pi_i$ in $D^{(r_i)}$ by means

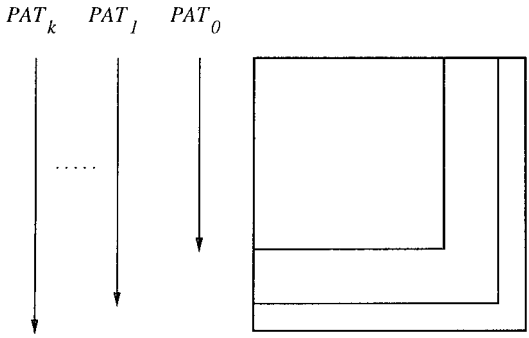

FIG. 9. The capsular matrices $PAT_0, PAT_1, ..., PAT_k$ in matrix PAT.

of the thread links set up during the construction of the index data structures and by branching in $D^{(r_i)}$ with π_i. We need $name(PAT_i)$ to do the latter task. When $i = k$, we have the locus of π. Unfortunately, a locus might have been removed in one of the refinement steps as it became a one-child node. This possibility makes the actual search procedure a little more complicated. It is described in Subsection 4.2 and requires $O(\log m)$ time with a single processor.

We now give the necessary details for the two parts in the pattern query.

4.1. *Part I: Computing Capsular Matrix Names*

We are given matrix PAT and want to compute the names of some of its submatrices of size $\Theta(m^2)$: the capsular matrices PAT_i. We compute each $name(PAT_i)$ as follows. We first process the whole matrix PAT in $O(\log m)$ time with $m^2/\log m$ processors in order to compute the names for some small submatrices in it of size $\Theta(\log m)$. We then apply the partial naming function from scratch as described in Subsection 2.2 to each matrix PAT_i except that we avoid processing the aforementioned small submatrices since we already have their names. We show that this requires $O(\log m)$ time with only $m^2/\log^2 m$ processors for each $name(PAT_i)$. In the following algorithms, if a pattern submatrix gets a name not previously assigned to a text submatrix, we immediately stop the pattern query and give a negative answer as PAT cannot occur in the text. From now on, we assume that this is not the case in order to simplify the presentation.

Roughly speaking, we divide PAT in three ways: horizontal stripes, vertical stripes, and diagonal stripes. These stripes contain some $\ell \times \ell$ submatrices of PAT, where ℓ is a power of two to be properly fixed. In the example shown in Fig. 10, we have picked $\ell = 4$. The three types of stripes are defined as follows:

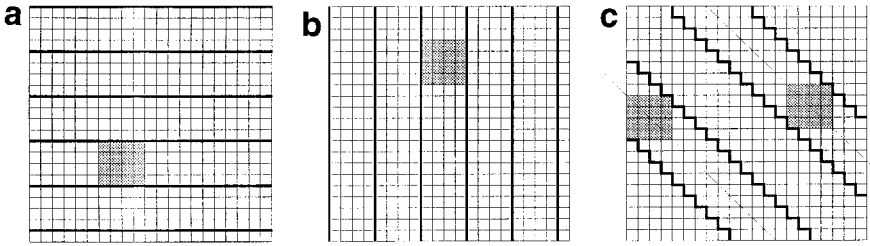

FIG. 10. The horizontal, vertical, and diagonal stripes of matrix PAT for $\ell = 4$. For the diagonal stripes, we only show two stripes that are not overlapping.

- $\lfloor m/\ell \rfloor$ disjoint *horizontal* stripes of height ℓ and width m, starting from the top of PAT (see Fig. 10a). They contain the $\ell \times \ell$ submatrices whose top leftmost entry (f, g) is such that $f-1$ is divisible by ℓ.
- $\lfloor m/\ell \rfloor$ disjoint *vertical* stripes of height m and width ℓ, starting from the left of PAT (see Fig. 10b). They contain the $\ell \times \ell$ submatrices whose top leftmost entry (f, g) is such that $g-1$ is divisible by ℓ.
- $2\lfloor m/\ell \rfloor - 1$ overlapping *diagonal* stripes. Each of them is centered around a diagonal and extends $\ell - 1$ diagonals to the left and to the right (see Fig. 10c). These stripes contain the $\ell \times \ell$ submatrices whose top leftmost entry (f, g) is such that $|f-g|$ is divisible by ℓ.

We find the names of the $\ell \times \ell$ submatrices in these stripes by traversing the tries $\mathscr{R}$ and $\mathscr{C}$ defined in Subsection 3.7. We recall that they store all text submatrices of side a power of two smaller than or equal to L (we therefore have to pick $\ell \leqslant L$). We only illustrate how to process the diagonal stripes (this task for the horizontal and vertical stripes is simpler and takes $O(m^2)$ work).

Processing of the Diagonal Stripes. We assign $\lfloor m/\ell \rfloor$ processors to each one of the $2\lfloor m/\ell \rfloor - 1$ diagonal stripes. Since we have $m^2/\log m$ processors, we need to pick ℓ as the largest power of two smaller than m such that ℓ is not too large (i.e., $\ell \leqslant L$) and we have enough processors (i.e., $2\lfloor m/\ell \rfloor^2 - \lfloor m/\ell \rfloor = O(m^2/\log m)$). As $L = O(\sqrt{\log n})$, we can fix a value of $\ell = \Theta(\sqrt{\log m})$ satisfying the above two conditions. Given a stripe, we use its $\lfloor m/\ell \rfloor$ processors to handle it independently of the others. We partition the $\ell \times \ell$ matrices contained in this stripe into $O(m/\ell)$ disjoint groups of adjacent matrices. Each group contains ℓ submatrices except possibly the last group, and forms a "staircase" (cf. Fig. 11).

The Task of a Processor. A single processor is assigned to each of those staircases and computes names for the matrices in it from the top leftmost to the bottom rightmost. In "descending" the staircase, the processor uses

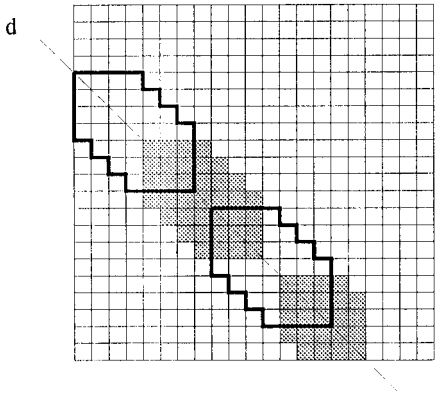

FIG. 11. The staircases along a diagonal stripe of matrix PAT for $\ell = 4$.

the overlap between adjacent matrices in it. Let B_1 be the top leftmost matrix in the staircase at hand and let B_2 be the next matrix in the staircase. The processor computes $row(B_1)$ and uses it to find its locus w in trie $\mathscr{R}$. If w does not exist or it is not marked, PAT does not occur in the text. Let us therefore assume that the processor succeeds in its task and so it retrieves $name(B_1)$ in w. Now we are in the situation depicted in Fig. 12a and want to "get" into the situation depicted in Fig. 12e. We proceed as follows. The processor follows the cross link out of w. This gives the locus $v \in \mathscr{C}$ of $col(B_1)$ (we go from Fig. 12a to 12b). After that, the processor takes the shortcut link from v to the locus $u \in \mathscr{C}$ of $col(B'')$, where B'' is matrix B_1 without its first column. Starting from u and using the column in PAT following B'', the processor finds the locus $w' \in \mathscr{C}$ of $col(C)$, where C is the matrix in boldface shown in Fig. 12c. The processor then follows the cross link out of w' and reaches the locus $v' \in \mathscr{R}$ of $row(C)$ (we go from Fig. 12c to 12d). By taking the shortcut link from v', the processor moves to the locus u' of $row(C')$, where C' is matrix C without its first row. Finally, starting out from u' and using the row in PAT below C', the processor finds the locus of $row(B_2)$ and so $name(B_2)$ for the next matrix B_2 in the staircase (we go from Fig. 12d to 12e). The above process of traversing a cross and a shortcut link, matching a column, traversing again a cross and shortcut link, and then matching a row is repeated until the processor has examined all the matrices in its staircase. If the processor fails in traversing a link or findings a name, PAT does not occur in the text.

We repeat a similar computation for the horizontal and vertical stripes. We store the names thus computed into an $m \times m$ array $SMALL$ initially set to zero. The name of a processed submatrix with top leftmost entry (f, g) in PAT is stored into $SMALL[f, g]$. The entries in $SMALL$ that are still zero at the end of the computation correspond to the submatrices in PAT not contained in the above stripes.

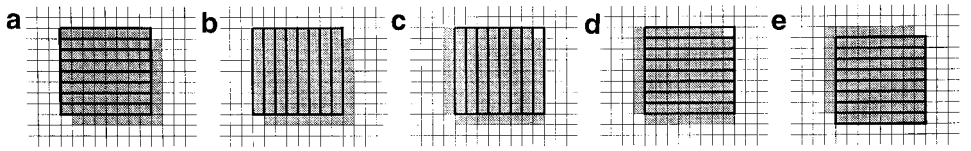

FIG. 12. The naming of two consecutive matrices in a staircase for $\ell = 8$.

LEMMA 4.1. *Computing the names for all the matrices in the horizontal, vertical, and diagonal stripes takes* $O(\log m)$ *time with* $m^2/\log m$ *processors.*

Proof. The diagonal stripes are the most expensive process. Their cost dominates the one for other stripes. Processing a staircase by a single processor takes $O(\ell^2) = O(\log m)$ time as $O(\ell^2)$ time is needed to find the name for the top leftmost matrix and $O(\ell)$ time is needed for each of the other $\ell - 1$ matrices (at most) in the staircase. As the $m^2/\log m$ processors are sufficient to process independently all the staircases in the stripes, we have a total cost of $O(\log m)$ time with $m^2/\log m$ processors. ∎

We now show how to compute each $name(PAT_i)$ in $O(m^2/\log m)$ work. The stripes play a crucial role in this name computation as we already have the names of the $\ell \times \ell$ submatrices processed in PAT_i.

LEMMA 4.2. *For any given capsular matrix* PAT_i, *it is possible to compute* $name(PAT_i)$ *by using only the horizontal, vertical, and diagonal stripes in* PAT.

Proof. We consider the case in which the $\ell_i \times \ell_i$ matrix PAT_i has a side ℓ_i which is not a power of two (the case in which ℓ_i is a power of two is analogous). We recall from Section 2 that computing $name(PAT_i)$ amounts to assigning names to its four covering matrices. We illustrate this situation in Fig. 13. Let TL_i, TR_i, BL_i, and BR_i denote such matrices. Let S be one of them. It is worth noting that S is of size $s \times s$, where s is a power of two such that $\ell \leqslant s \leqslant \ell_i \leqslant m$. We divide S into $(s/\ell)^2$ submatrices of size $\ell \times \ell$ each (s is divisible by ℓ as both are powers of two). These submatrices

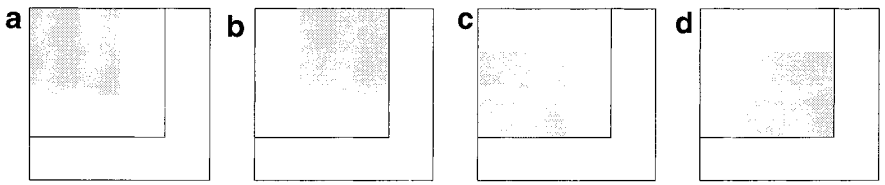

FIG. 13. A capsular matrix PAT_i and its four covering matrices: (a) TL_i, (b) TR_i, (c) BL_i, and (d) BR_i.

are those obtained while computing the names. We show that they are contained in the stripes of PAT. The proof is by case analysis.

Case $S = TR_i$. The first row in S is part of the first row in PAT. Let us examine the $\ell \times \ell$ matrices partitioning S: The first row of each of them is aligned with row f of PAT, for a value of $f \in \{1, \ell+1, 2\ell+1, ..., s-\ell+1\}$. That is, the top leftmost entry corresponds to an entry (f, g) in PAT, such that $f-1$ is divisible by ℓ, and so these matrices are surely contained in the horizontal stripes. The case in which $S = BL_i$ is similar to the one just discussed, except that the first column in S is part of the first column in PAT. Consequently, the $\ell \times \ell$ matrices are contained in the vertical stripes. The case in which $S = TL_i$ is analogous to either of the two cases above.

Case $S = BR_i$. The main diagonal of S is on the main diagonal of PAT by the definition of PAT_i and BR_i. The main diagonal in each of the $\ell \times \ell$ matrices partitioning S must be aligned with diagonal d of PAT, where $|d| \in \{0, \ell, 2\ell, ..., s-\ell\}$. That is, the top leftmost entry corresponds to an entry (f, g) in PAT, such that $d = f-g$ is divisible by ℓ, and so these matrices are contained in the diagonal stripes. ▮

Lemma 4.2 allows us to conclude that it is possible to compute $name(PAT_i)$ starting from the names stored in array $SMALL$. The algorithm computing $name(PAT_i)$ is a simple modification of the one computing the partial naming function from scratch as mentioned Subsection 2.2: compute the name of the matrix recursively; when the name of an $\ell \times \ell$ matrix has to be found, make a table look-up in $SMALL$ instead of going on with the recursion. We show that the computation of each $name(PAT_i)$ takes $O(\log m)$ time with $m^2/\log^2 m$ processors and the total cost for all capsular matrices PAT_i is $O(\log m)$ time with $m^2/\log m$ processors.

THEOREM 4.3. *The computation of* $name(PAT_i)$, *for* $0 \leqslant i \leqslant k$, *takes a total of* $O(\log m)$ *time with* $m^2/\log m$ *processors.*

Proof. The correctness of our approach follows from Lemma 4.2. As for the time analysis, the initialization of array $SMALL$ and its setup can be done in $O(\log m)$ time with $m^2/\log m$ processors as a consequence of Lemma 4.1. The computation of each $name(PAT_i)$ yields a quaternary tree of recursive calls. This tree has $O((s/\ell)^2) = O(m^2/\log m)$ nodes and height $O(\log s - \log \ell + 1) = O(\log m)$. The tree computation can be therefore executed in $O(\log m)$ time with $m^2/\log^2 m$ processors [37]. We perform these computations in parallel for each i, $0 \leqslant i \leqslant k$, and therefore the total running time is $O(\log m)$ with $m^2/\log m$ processors. ▮

4.2. *Part II*: *Traversing the Refinement Trees*

The search in the refinement trees is intuitively simple. It works by induction on $i = 0, ..., k$ and uses the decomposition of $\pi = \pi_0 \pi_1 \cdots \pi_k$. Let pat_i denote the Lstring $\pi_0 \cdots \pi_i$ and pat_{-1} denote the empty Lstring whose locus is the root of refinement tree $D^{(r_0)}$. We start out at $root(D^{(r_0)})$ with $i = 0$. Let us assume that we have found the locus $\hat{u}$ of pat_{i-1} in $D^{(r_i)}$. We branch from $\hat{u}$ with π_i to find the locus u of pat_i in $D^{(r_i)}$ as $pat_i = pat_{i-1}\pi_i$. If u exists, we locate its copy in $D^{(r_{i+1})}$ by means of the thread links and iterate with $\hat{u} := u$ and $i := i + 1$. In the last iteration k, we reach the locus of $pat_k = \pi$. Unfortunately, u might have become one-child in a refinement tree and thus removed. This possibility makes the actual search procedure a little more complicated than what we have just outlined.

ALGORITHM TRAVERSE. We assume that $name(PAT_i)$ is available, for $0 \leqslant i \leqslant k$. We set $i := 0$, $\hat{u} := root(D^{(r_0)})$, and repeat Steps 1 and 2 below until we get an answer:

Step 1. The input is the locus $\hat{u}$ of pat_{i-1} in $D^{(r_i)}$. We branch from $\hat{u}$ with π_i to reach the extended locus u of pat_i (we do this by following the pointer in position $name(PAT_i)$ in the array $OUT(\hat{u})$ storing the outgoing arcs of $\hat{u}$). If u does not exist, we stop the search with a negative answer. Otherwise, let β_u be the chunk labeling u. We examine two cases:

- β_u is equal to π_i. If $i = k$, we stop with a positive answer. If $i < k$, we go on and execute Step 2.
- β_u is longer than π_i. Let (h, j, p, q) be the descriptor of β_u. We check to see if PAT equals the $m \times m$ submatrix of A whose top leftmost entry is (h, j). We return the check result and stop.

Step 2. The input is the locus u of pat_i in $D^{(r_i)}$. We follow the thread link from $u \in D^{(r+1)}$ to its copy in $D^{(r)}$ for the decreasing sequence of values $r = r_i - 1, r_i - 2, ..., r_{i+1}$. If we reach $r = r_{i+1}$ and u exists in $D^{(r)}$, we set $\hat{u} := u$, $i := i + 1$, and execute Step 1. Otherwise, $u \in D^{(r+1)}$ but $u \notin D^{(r)}$ and so the thread link goes from $u \in D^{(r+1)}$ to another node $w \in D^{(r)}$. We have two cases:

- $r = r_{i+1}$. We check to see if w is the locus of pat_{i+1}. If the check result is false, we give a negative answer. If the check is true and $i + 1 = k$, we give a positive answer. (In both cases, we stop the search.) Finally, if the check is true and $i + 1 < k$, we set $i := i + 1$, $u := w$, and repeat Step 2.

- $r > r_{i+1}$. Let (h, j, p, q) be the descriptor of β_w. We proceed as in Step 1: We compare PAT to the $m \times m$ submatrix of A whose top leftmost entry is (h, j) to return the check result and stop.

THEOREM 4.4. *Let us assume that the index data structures for matrix A are produced and that name(PAT_i) is available, for $0 \leqslant i \leqslant k$. Algorithm Traverse checks whether PAT occurs in A in $O(\log m)$ time with a single processor.*

Proof. We first discuss the efficient implementation and the correctness of Steps 1–2 in Algorithm Traverse. Then we analyze the complexity. We use induction for the correctness, in which the initial computation before Steps 1–2 is the base. Let us therefore assume that we have reached by induction either node $\hat{u}$ before executing Step 1 or node u before executing Step 2.

Let us examine Step 1: All the Lstrings whose prefix is pat_{i-1} must descend from $\hat{u}$ as it is its (unique) locus. Branching from $\hat{u}$ is implemented by accessing position $name(PAT_i)$ in $OUT(\hat{u})$ as we exploit the fact that the names are consistent. Consequently, if we fail to branch from $\hat{u}$, no pattern occurrence exists. If we instead reach u, we know that π_i is a prefix of β_u. We have two cases: (i) $\beta_u = \pi_i$ and (ii) $|\beta_u| > |\pi_i|$. In case (i), u is the locus of $pat_i = pat_{i-1}\pi_i = pat_{i-1}\beta_u$. If $i = k$, we have found a pattern occurrence (just take any descending leaf from u); otherwise, we keep the induction for Step 2. In case (ii), we use Lemma 3.4 to infer that $|\beta_u| \geqslant 2^{r_i+1}$ as it cannot be $|\beta_u| = 2^{r_i}$ (the former case was already discussed). But then β_u is longer than $\pi_i \cdots \pi_k$ as the latter string is shorter than 2^{r_i+1} by the fact that $\pi_i, ..., \pi_k$ are "maximal" clunks of a power-of-two length. Consequently, u is the only candidate for being the extended locus of $\pi = pat_{i-1}\pi_i \cdots \pi_k$ in the refinement tree. This and the fact that leaf (h, j) descends from u (by Condition (D2) on the refinement trees) motivate the equality check on PAT.

Let us examine Step 2: All the Lstrings whose prefix is pat_i must descend from u. If u exists also in $D^{(r_{i+1})}$, we keep the induction for Step 1. We verify this condition by traversing the thread links. Otherwise, we stop traversing at tree $D^{(r)}$. There we find node w, which was the only child of u in $\tilde{D}^{(r+1)}$ before its removal as one-child node during the refinement to get $D^{(r)}$. We infer that w is the locus of an Lstring α that has prefix pat_i and length $|pat_i| + 2^r$, where 2^r is due to the refiner 2^r previously applied to $D^{(r+1)}$. We have two cases: (I) $r = r_{i+1}$ and (II) $r > r_{i+1}$. In case (I), we have that w is the only candidate for being the locus of pat_{i+1} as α has prefix pat_i and length $|pat_i| + 2^{r_{i+1}} = |pat_{i+1}|$ whereas the Lstring having locus in $parent(w)$ is shorter than pat_i. We therefore need to compare α to pat_{i+1} to see if w is its locus, i.e., we compare the matrix represented by α and

PAT_{i+1} through their names. As a result, we can either give an answer and stop or we can keep the induction for Step 2 (when $i+1<k$). We have to discuss case (II). We know that Lstring α has length $|pat_i|+2^r \geqslant |pat_i|+2^{r_{i+1}+1} > |pat_i|+|\pi_{i+1}\cdots\pi_k|$. As $\pi = pat_i\pi_{i+1}\cdots\pi_k$, we derive that α is longer than π. Consequently, w is the only candidate for being the extended locus of π in the refinement tree and we proceed as in Step 1.

In order to analyze the complexity of the algorithm, we wish to point out that each operation in Steps 1–2 takes $O(1)$ time with a single processor by assuming that we can use names to compare the capsular matrices (we also remark that $PAT_k = PAT$ so that comparing PAT takes $O(1)$ time). We therefore have to evaluate the number of total operations. Let us take iteration i into consideration. Step 1 performs $O(1)$ operations and then either it stops or goes to Step 2. Step 2 performs no more than $O(r_i - r_{i+1})$ operations (due to traversing the thread links). If it does not stop, it goes to Step 1 or executes again Step 2. In both cases, it increments i by one. This means that Steps 1 and 2 cannot be executed more than $O(k)$ times. The total cost is therefore bounded by $O(k+\sum_{i=0}^{k-1}(r_i - r_{i+1})) = O(k + r_0 - rk) = O(\log m)$ time with a single processor. ▌

We can finally state our result on the two-dimensional on-line pattern matching problem.

THEOREM 4.5. *Let A be an $n\times n$ text matrix and PAT be an $m\times m$ pattern matrix, given on-line for pattern matching purposes. We can build the Index data structures for A in $O(\log n)$ time with n^2 processors so that we can check to see if any PAT occurs in A in $O(\log m)$ time with $m^2/\log m$ processors. The computation is performed on an Arbitrary CRCW PRAM.*

Proof. We build the data structures for A in $O(\log n)$ time and n^2 processors by Theorem 3.10. Given PAT, we compute $name(PAT_i)$, for $0 \leqslant i \leqslant k$, in $O(\log m)$ time with $m^2/\log m$ processors by Theorem 4.3. We then search in the refinement trees in $O(\log m)$ time with a single processor by Theorem 4.4.

5. CONCLUSIONS

We have shown how to build the index data structures for an $n\times n$ text matrix A in $O(\log n)$ time with n^2 processors with applications to on-line pattern matching and to problems related to the gathering of statistical information about the matrix A. The query algorithm is work optimal and the construction algorithm is work optimal only for arbitrary and large alphabets. It would be interesting to obtain an $O(n^2)$ work construction algorithm for a small alphabet, still having optimal work queries.

REFERENCES

1. A. Amir, G. Benson, and M. Farach, Optimal parallel two-dimensional pattern matching, *in* "Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures," pp. 79–85, Assoc. Comput. Mach., New York, 1993.
2. A. Amir and M. Farach, Two-dimensional dictionary matching, *Inform. Process. Lett.* **44** (1992), 233–239.
3. A. Amir, M. Farach, and Y. Matias, Efficient randomized dictionary matching algorithms, *in* "Proc. 3rd Combinatorial Pattern Matching Conference," Lecture Notes in Computer Science, Vol. 644, pp. 259–272, Springer-Verlag, New York/Berlin, 1992.
4. A. Amir and G. M. Landau, Fast parallel and serial multi-dimensional approximate array matching, *in* "Sequences: Combinatorics, Compression; Security and Transmission" (R. M. Capocelli, Ed.), pp. 120–130, Springer-Verlag, Berlin, 1990.
5. A. Apostolico, The myriad virtues of subword trees, *in* "Combinatorial Algorithms on Words" (A. Apostolico and Z. Galil, Eds.), pp. 85–95, Springer-Verlag, Berlin, 1985.
6. A. Apostolico and Z. Galil (Eds.), "Combinatorial Algorithms on Words," Springer-Verlag, Berlin, 1985.
7. A. Apostolico and Z. Galil (Eds.), "Pattern Matching Algorithms," Oxford Univ. Press, New York, 1997.
8. A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin, Parallel construction of a suffix tree with applications, *Algorithmica* **3** (1988), 347–365.
9. O. Berkman and U. Vishkin, Finding level ancestors in trees, *J. Comput. System Sci.* **48** (1994), 214–230.
10. P. C. B. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena, Improved deterministic parallel integer sorting, *Inform. and Control* **94** (1991), 29–47.
11. S. K. Chang, Q. Y. Shi, and C. W. Yan, Iconic indexing by 2-D strings, *IEEE Trans. Pattern Anal. Mach. Intelligence* **9** (1987), 413–428.
12. R. Cole, Parallel merge sort, *SIAM J. Comput.* **17** (1988), 770–785.
13. R. Cole, M. Crochemore, Z. Galil, L. Gasieniec, R. Hariharan, S. Muthuhrishnan, K. Park, and W. Rytter, Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions, *in* "Proc. 34th Symposium on Foundations of Computer Science," pp. 248–258, IEEE Press, New York, 1993.
14. R. Cole and U. Vishkin, Approximate parallel scheduling. Part I. The basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* **17** (1988), 128–142.
15. R. Cole and U. Vishkin, Faster optimal parallel prefix sums and list ranking, *Inform. and Comput.* **81** (1989), 334–352.
16. M. Crochemore and W. Rytter, Usefulness of the Karp–Miller–Rosenberg algorithm in parallel computations on strings and arrays, *Theoret. Comput. Sci.* **88** (1991), 59–82.
17. M. Crochemore and W. Rytter, "Text Algorithms," Oxford Univ. Press, New York, 1994.
18. A. Czumaj, Z. Galil, L. Gąsieniec, K. Park, and W. Plandowski, Work-time-optimal parallel algorithms for string problems, *in* "Proc. 27th Symposium on Theory of Computing," pp. 713–722, Assoc. Comput. Mach., New York, 1995.
19. M. Farach and S. Muthukrishnan, Optimal parallel dictionary matching and compression, *in* "Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures," pp. 244–253, Assoc. Comput. Mach., New York, 1995.

20. M. Farach and S. Muthukrishnan, Optimal logarithmic time randomized suffix tree construction, *in* "Proc. 23rd International Colloquium on Automata, Languages and Programming," Lecture Notes in Computer Science, Vol. 1099, pp. 550–561, Springer-Verlag, New York/Berlin, 1996.
21. P. Ferragina and F. Luccio, On the parallel dynamic dictionary matching problem: New results with application, *in* "Proc. 4th European Symposium on Algorithms," Lecture Notes in Computer Science, Vol. 1136, pp. 261–275, Springer-Verlag, New York/Berlin, 1996.
22. M. J. Fischer and L. Ladner, Parallel prefix computation, *J. Assoc. Comput. Mach.* **27** (1980), 831–838.
23. R. Giancarlo, A generalization of the suffix tree to square matrices, with applications, *SIAM J. Comput.* **24** (1995), 520–562.
24. R. Giancarlo, An index data structure for matrices, with applications to fast two-dimensional pattern matching, *in* "Proc. of Workshop on Algorithms and Data Structures," Lecture Notes in Computer Science, Vol. 709, pp. 337–348, Springer-Verlag, New York/Berlin, 1993.
25. R. Giancarlo and R. Grossi, Parallel construction and query of suffix trees for two-dimensional matrices, *in* "Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures," pp. 86–97, Assoc. Comput. Mach., New York, 1993.
26. R. Giancarlo and R. Grossi, "Parallel Construction and Query of Suffix Trees for Square Matrices," Bell Laboratories Technical Report, 11272-931015-26, 1993.
27. R. Giancarlo and R. Grossi, Multi-dimensional pattern matching with dimensional wildcards: Data structures and optimal on-line search algorithms, *J. Algorithms* **24** (1997), 223–265.
28. R. Giancarlo and R. Grossi, On the construction of classes of suffix trees for square matrices: Algorithms and applications, *Inform. and Comput.* **130** (1996), 151–182.
29. G. H. Gonnet, "Efficient Searching of Text and Pictures," Technical Report, OED-88-02, University of Waterloo, 1988.
30. D. Gusfield, "Algorithms on Strings, Trees, and Sequences," Cambridge Univ. Press, New York, 1997.
31. T. Hagerup, On saving space in parallel computation, *Inform. Process. Lett.* **29** (1988), 327–329.
32. R. Hariharan, Optimal parallel suffix tree construction, *J. Comput. Sci.* **55** (1997), 44–69.
33. L. Hui, Color set size problem with applications to string matching, *in* "Proc. 3rd Combinatorial Pattern Matching Conference," Lecture Notes in Computer Science, Vol. 644, pp. 227–240, Springer-Verlag, New York/Berlin, 1995.
34. R. Jain, "Workshop Report on Visual Information Systems," Technical Report, National Science Foundation, 1992.
35. J. JáJá, "An Introduction to Parallel Algorithms," Addison–Wesley, Reading, MA, 1992.
36. R. Karp, R. Miller, and A. Rosenberg, Rapid identification of repeated patterns in strings, arrays and trees, *in* "Proc. 4th Symposium on Theory of Computing," pp. 125–136, Assoc. Comput. Mach., New York, 1972.
37. R. M. Karp and V. Ramachandran, Parallel algorithms for shared-memory machines, *in* "Handbook of Theoretical Computer Sciences" (J. van Leeuwen, Ed.), Vol. A, pp. 869–941, Elsevier/MIT Press, Cambridge, MA, 1990.
38. Z. M. Kedem, G. M. Landau, and K. V. Palem, Parallel suffix-prefix-matching algorithm and applications, *SIAM J. Comput.* **25** (1996), 998–1023.

39. D. K. Kim, Y. K. Kim, and K. Park, Constructing suffix arrays for multi-dimensional matrices, *in* "Proc. 9th Combinatorial Pattern Matching Conference," Lecture Notes in Computer Science, Vol. 1448, pp. 126–139, Springer-Verlag, New York/Berlin, 1998.
40. D. E. Knuth, "The Art of Computer Programming. Vol. 3. Sorting and Searching," Addison–Wesley, Reading, MA, 1973.
41. D. E. Knuth, J. H. Morris, and V. B. Pratt, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977), 189–195.
42. A. Lempel and J. Ziv, Compression of two-dimensional images, *in* "Combinatorial Algorithms on Words" (A. Apostolico and Z. Galil, Eds.), NATO ASI Series F, Vol. 12, pp. 141–154, Springer-Verlag, Berlin, 1984.
43. U. Manber and G. Myers, Suffix arrays: A new method for on-line string searches, *SIAM J. Comput.* **22** (1993), 935–948.
44. T. R. Mathies, "A Fast Parallel Algorithm to Determine Edit Distance," Technical Report, TR CMU-CS, 1988.
45. Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv, Augmenting suffix trees with applications, *in* "Proc. 6th European Symposium on Algorithms," Lecture Notes in Computer Science, Vol. 1461, pp. 67–78, Springer-Verlag, New York/Berlin, 1998.
46. E. M. McCreight, A space economical suffix tree construction algorithm, *J. Assoc. Comput. Mach.* **23** (1976), 262–272.
47. D. R. Morrison, PATRICIA—Practical algorithm to retrieve information coded in alphanumeric, *J. Assoc. Comput. Mach.* **15** (1968), 514–534.
48. S. Muthukrishnan and K. Palem, Highly efficient dictionary matching in parallel, *in* "Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures," pp. 69–78, Assoc. Comput. Mach., New York, 1993.
49. S. C. Sahinalp and U. Vishkin, Symmetry breaking for suffix tree construction, *in* "Proc. 26th Symposium on Theory of Computing," pp. 300–309, Assoc. Comput. Mach., New York, 1994.
50. D. Sheinwald, A. Lempel, and J. Ziv, Compression of pictures by finite state encoders, *in* "Sequences: Combinatorics, Compression, Security and Transmission" (R. M. Capocelli, Ed.), pp. 326–347, Springer-Verlag, Berlin, 1990.
51. Y. Shiloach and U. Vishkin, Finding the maximum, merging and sorting in a parallel computation model, *J. Algorithms* **2** (1981), 88–102.
52. J. A. Storer, Lossy on-line dynamic data compression, *in* "Sequences: Combinatorics, Compression, Security and Transmission" (R. M. Capocelli, Ed.), pp. 348–357, Springer-Verlag, Berlin, 1990.
53. J. A. Storer, Lossless image compression using generalized LZ1-type methods, *in* "Proc. Data Compression Conference," pp. 290–299, IEEE, New York, 1996.
54. R. E. Tarjan and U. Vishkin, An efficient parallel biconnetivity algorithm, *SIAM J. Comput.* **14** (1985), 862–874.
55. P. Weiner, Linear pattern matching algorithm, *in* "Proc. 14th IEEE Symp. on Switching and Automata Theory," pp. 1–11, IEEE, New York, 1973.