SIAM J. COMPUT.
Vol. 20, No. 1, pp. 22-40, February 1991

DETERMINISTIC SAMPLING—A NEW TECHNIQUE FOR FAST PATTERN MATCHING*

UZI VISHKIN†

Abstract. Consider the following three-stage strategy for recognizing patterns in larger scenes:
Mimic randomization deterministically. Sample several positions of the pattern.
Search for sample. Find all occurrences of the sample in the scene.
Verify. For each occurrence of the sample, verify occurrence of the full pattern.

This strategy has led to the core of the new idea given in this paper. Consider the string matching problem. Given the pattern, a sample of its positions is carefully selected whose size is at most logarithmic (the *deterministic sample*). Then, the sample is searched for. For nonperiodic patterns, the sample has the following perhaps surprising property. It is possible to disqualify all occurrences of the sample positions but one, within each "neighborhood" of locations in the text, without any further comparisons of characters. This provides *sparse* verification.

This approach enables the text analysis (stages "search for sample" and "verify") to be performed in $O(\log^* n)$ time and optimal speedup on a PRAM. This improves on the previous fastest optimal speedup result. It also leads to a new serial algorithm for string matching that runs in linear time including preprocessing.

The approach is expected to be applicable for pragmatic pattern recognition problems.

In some sense the algorithms are based on degenerate forms of computation, such as AND and OR of a large number of bits. However, traditional machine designs do not take advantage of such degeneracies, and usual complexity measures do not even enable them to be reflected. This leads to the conclusion of the paper with some speculative thoughts on desirable capabilities that would enhance computing machinery for some pattern recognition applications.

Key words. string matching, serial algorithms, parallel algorithms, deterministic sampling

AMS(MOS) subject classifications. 68P99, 68Q20, 68T10, 68Q10

1. Introduction. Suppose we are given a string of length n, $T[1 \cdots n]$, called the *text*, and a shorter string of length m, $P[1 \cdots m]$, called the *pattern*. The *string matching* problem is to find all "*starting*" locations $1 \leqq i \leqq n-m+1$ in the text, such that the pattern matches character by character the substring of the text $T[i, i+1, \cdots, i+m-1]$. As stated in [Ga85b], this is one of the most extensively studied problems in theoretical computer science.

The naive algorithm for the problem is as follows. Test whether each location $i = 1, 2, \cdots, n-m+1$ is a starting location by m character-by-character comparisons. This totals $O(nm)$ operations, or $O(1)$ time using nm processors on a CRCW PRAM. Nontrivial algorithms for this problem consist of two stages. In the first stage, the "*pattern analysis*," they construct a table based on analysis of the pattern only. In the second and final stage, the "*text analysis*," the text is analyzed. The table built in the first stage helps to minimize repeated reading of the same text characters.

There are several serial algorithms for the string matching problem: by Knuth, Morris, and Pratt [KMP77] (and the heuristic improvement by Boyer and Moore [BM77]), the randomized algorithm by Karp and Rabin [KR87], the real-time algorithm using a constant number of registers by Galil and Seiferas [GS83], and a serial

* Received by the editors August 30, 1989; accepted for publication (in revised form) March 23, 1990. This research was supported by National Science Foundation grants CCR-8615337 and CCR-8906949 and Office of Naval Research grant N00014-85-K-0046.

† Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742; and Department of Computer Science, Tel Aviv University, Tel Aviv, Israel.

simulation of the parallel algorithm by Vishkin [Vi85]. The first contribution concerning efficient parallel string matching was by Galil [Ga85a], where a framework benefiting from periodicity properties in strings was introduced. Similar properties were used in later parallel string matching algorithms. The algorithm in Galil's original paper runs in logarithmic time and is optimal for an alphabet whose size is fixed. Vishkin [Vi85] proposed a new idea that has led to an optimal speedup algorithm regardless of the alphabet size. A recent paper by Breslauer and Galil [BG88] added the following surprising perspective to our work. They observed that the new idea from [Vi85] implies that the string matching problem is not more difficult, from the parallel algorithmic point of view, than the problem of finding the maximum among n elements. This made possible a doubly logarithmic optimal parallel algorithm for the problem. In [KR87], Karp and Rabin present an optimal logarithmic parallel implementation of their randomized algorithm. Kendem, Landau, and Palem [KLP89] recently gave another parallel algorithm. Finally, we refer the reader to a survey on string problems by Galil [Ga85b].

Our main results include:

(1) A new linear time serial algorithm for the string matching problem.

(2) A new text analysis parallel algorithm that runs in $O(\log^* n)$ time using an optimal number of processors.

(3) The text analysis algorithm is based on a pattern analysis stage that takes $O(\log^2 m/\log\log m)$ time using an optimal number of processors.

(4) A randomized implementation of the pattern analysis needs $O(\log m)$ time, with high probability, using an optimal number of processors. Using the output of the randomized implementation, all text analysis results carry through (as deterministic results).

The deterministic sampling idea. All algorithms in the present paper rely on the following core idea. Given a nonperiodic pattern, our pattern analysis stage constructs a small "*deterministic sample* (*denoted DS*)" of pattern positions. This sample is an ordered set of size $l \leqq \log m - 1$. Specifically, $DS = [ds(1), ds(2), \cdots, ds(l)]$, where each $ds(j)$, $1 \leqq j \leqq l$, is a different integer between 1 and m. The main step of our *basic* text analysis tests whether each location $i = 1, 2, \cdots, n - m + 1$ can be a starting location by l comparisons with the sample pattern positions. Some locations of the text will pass this test and some will fail, and therefore be disqualified as starting locations. A perhaps surprising property of DS implies that there is a way for drastically disqualifying at once (i.e., simultaneously, in one parallel round) additional locations in the text, so that any remaining nondisqualified location is *unique* in some successive substring of length $m/2$.

Theoretically, the deterministic sampling idea can be viewed as getting a "signature" of the pattern by using a small sample of its locations. Concise signatures are natural for randomized algorithms as shown in the algorithm of [KR87]. We selected the name *deterministic sampling* to convey the possibility of getting signatures using deterministic means. Interestingly, the Karp–Rabin signature concept does not seem to be less involved since it blends all entries of the pattern rather than samples a few positions of the pattern. Our randomized parallel version compares favorably with theirs: The pattern analysis result is logarithmic time and optimal speedup, with high probability, in both papers. However, while the Karp–Rabin text analysis result is randomized and logarithmic time (with high probability), ours is deterministic and $O(\log^* n)$ time; both results achieve optimal speedup. Randomized algorithmics, as advocated in Rabin [Ra76], is an appealing concept. Our paper follows [A78], [BR89],

[CV86], [Lu88], and [MNN89] in demonstrating another angle of this concept. The deterministic sample idea shows how a randomized way of thinking can enrich the design of deterministic algorithms.

Pattern recognition based on small samples is apparently an intuitive idea. Our contribution in this respect can be summarized as presenting the first deterministic string matching algorithms that are guided by this idea, and whose worst-case performance is provably efficient. The literature records works in this direction in the 1950s. We mention one and refer the interested reader to references therein. Suppose we are given i pattern strings and a single target string, each of length m where $i \ll m$. The problem is to find whether one of the pattern strings matches the target string. A simple observation in [Gi59] is that it is enough to read at most i positions of the target string in order to disqualify all pattern strings, but one, as possible matches for the target string. Our work relates also to the heuristic of [BM77]. They used a single "most notable" character for speeding up the algorithm of [KMP77], however, there was no guarantee that such a character would always be very helpful. Our construction can be phrased as picking a set of at most $\log m - 1$ notable characters, which is provably helpful.

Our parallel pattern analysis algorithm is slower than the one in [BG88]. However, our text analysis algorithm is faster. It is not hard to imagine instances where the pattern is available in advance and there is no pressure to process it very fast, while it is important to process the text as fast as possible. Using such justification, [U85] gave an interesting serial algorithm for an approximate string matching problem whose text analysis takes linear time, but the pattern analysis might even need exponential time. Recall that [BG88] showed that the string matching problem is not more difficult than finding the maximum among n elements. Since [Va75] showed that n processors need $\Omega(\log \log n)$ time to find the maximum among n elements on a parallel comparison model of computation, it is interesting to phrase our text analysis result as follows: assuming some preprocessing of the pattern, the text analysis problem is actually easier than finding the maximum among n elements.

There is a remarkably small number of problems for which there exist optimal parallel algorithms that run in sub-doubly-logarithmic time (i.e., $o(\log \log n)$ time). Constant time optimal parallel algorithms include: (a) OR and AND of n bits; (b) finding the minimum among n elements where the input consists of integers in the domain $[1, \cdots, n^c]$ (see [FRW88]); (c) $\log n$-coloring of a cycle, [CV86]; (d) some probabilistic computational geometric problems [St88]. A data-structure that provides for optimal $O(\log^* n)$ time and even inverse-Ackermann time parallel algorithms for some problems on trees and arrays, assuming some preprocessing, is given in [BV89]. Sub-doubly-logarithmic merging algorithms (on a CREW PRAM) were recently given in [BV90]. In [BV89], Berkman and Vishkin explain why constant-time and optimal speedup represents an ultimate theoretical goal for designers of parallel algorithms. Since for almost any interesting problem this goal is (provably) unachievable, any result that approaches this goal, such as our text analysis algorithm, is somewhat surprising.

Further applicability. We hope that the deterministic sample idea will find other applications in the pattern matching area. Our results extend to string matching in higher dimensions and approximate string matching if some assumptions are made about the pattern. The main difficulty we had in obtaining more general analytic results is that the concept of periodicity becomes vague already for two dimensions. The flow of our algorithms is quite rigid, once the deterministic sample is fixed. This invites

research for extending our algorithms to more specialized computer architectures. In the last section several alternative implementations of our ideas are considered. One of them is serial and is likely to need less than linear time in practice. The second suggests a reasonable assumption about the pattern that makes possible extension to higher dimensions. The third suggests an assumption about the pattern that makes possible extension to approximate matches. It might be relevant for some image processing applications. The fourth item is more speculative, as it suggests reconsidering some standard complexity measures under some circumstances.

The model of parallel computation that is needed for this paper is the *common concurrent-read concurrent-write* (CRCW) *parallel random access machine* (PRAM). A PRAM employs p synchronous processors all having access to a common memory. The common CRCW PRAM allows several processors to write simultaneously into the same memory location, provided that they try to write the same value. For convenience, however, we will describe our algorithms for the slightly more powerful *priority* CRCW PRAM; in case several processors attempt to write simultaneously into the same memory location, the one with the smallest index succeeds. Fortunately, all uses of the priority concurrent-write assumption are in order to solve the same problem. The next section states the problem and quotes the standard way for solving it on a common CRCW without asymptotic loss of efficiency. We comment on formulation of parallel complexity results in the present paper. While a bound of the form T time using p processors can always be stated as $O(T)$ time and (a total of) $O(pT)$ operations, the converse will also be true throughout this paper (but not in general). That is, T time and X operations will mean $O(T)$ time using X/T processors.

The paper is organized as follows. Section 3 presents the pattern analysis, and § 4 presents two basic text analyses. One runs in constant time and the other uses a linear number of operations, and thereby provides a linear time serial algorithm. Section 5 combines the two algorithms into an optimal $O(\log^* n)$ time text analysis, and § 6 concludes the paper.

For fast understanding of the main ideas, we suggest figuring out the definition of WITNESS in § 2, and the definition of the auxiliary column sample problem, its computation, and the deterministic sample in § 3. In § 4, understand the basic constant-time text analysis (including the Ricochet property). Then proceed to the basic optimal speedup algorithm. Understand how the serialization in advancing through the deterministic sample helps to reduce the total number of operations. In § 5, the main idea is in Stage 2. Understanding the input (and thereby the output) for each iteration should suffice.

2. Preliminaries.

Periodicity in strings. The main insight in Galil's [Ga85a] parallel string matching algorithm was to use the notion of periods in strings. We refer the reader to that paper for proofs of the facts stated below. Let u and w be two strings. u is a *period* of w if w is a prefix of u^k for some integer k, or equivalently if w is a prefix of uw. (This equivalence of definitions for *period* is called the *equivalence fact.*) The shortest period of a string w is the *period* of w. w is *periodic* if the length of its period is at most half its length; otherwise, it is nonperiodic.

The conflicting occurrences fact. Consider a pattern w whose period is u. Suppose w occurs at position i of some text string. Then it is impossible to have another occurrence of w at location j, for $i<j<i+|u|$.

The nonperiodic prefix fact. If the pattern is periodic (let p be the length of the period), then the prefix of the pattern of length $2p-1$ must be nonperiodic.

The fundamental observation regarding periodicity of strings, from which the above facts can be derived, is called the *gcd lemma* [LS62]: If w has two periods of length p and q, where $|w| \geqq p+q$, then w must have a period of length gcd (p, q).

Array WITNESS. Consider a nonperiodic pattern $P[1, \cdots, m]$. For any index i, $1 < i \leqq m/2$ consider laying two copies of the pattern one above the other where the first symbol of the upper copy aligns above the ith symbol of the lower copy, as in the example below, and the prefix $P[1, \cdots, m-i+1]$ of the upper copy aligns over the suffix $P[i, \cdots, m]$ of the lower copy. By the conflicting occurrences fact these prefix and suffix must be different. This means that for at least one $1 \leqq k \leqq m-i+1$, $P(k) \neq P(i-1+k)$. WITNESS (i) is one such index k (where $1 < i \leqq m/2$).

Example. Let the pattern be $P[1, \cdots, 7] = ababbaa$. This is a nonperiodic pattern and the suffix $P[3, 4, 5, 6, 7] = abbaa$ differs from the prefix $P[1, 2, 3, 4, 5] = ababb$ in the last three positions, see as below:

$$a\ b\ a\ b\ b\ a\ a$$

$$a\ b\ a\ b\ b\ a\ a$$

Therefore WITNESS (3) could be either 3 or 4 or 5, representing the "columns" in which the two copies of the pattern differ.

Comment. For the present paper we need only this definition of WITNESS. We briefly relate this definition to the discussion of the papers by [Vi85] and [BG88] in the Introduction. The new idea in [Vi85] was to use the information in array WITNESS for a very powerful mechanism (called *duel*): Suppose two candidate locations j and $j+i-1$ (in the text) whose distance i is small enough (i.e., $1 < i \leqq m/2-1$) are given. The duel mechanism enables us to eliminate at least one of these two candidates based on the contents of WITNESS (i). The reader is referred to [Vi85] for more information. Breslauer and Galil [BG88] have observed that application of the duel mechanism (as part of a string matching algorithm) and elimination of the smaller among two elements (as part of an algorithm for finding the maximum among n elements) lead to similar outcomes.

Reducing the periodic case to the nonperiodic case.

LEMMA 2.1. *Suppose we know that the pattern is periodic and can be presented as $u^k v$, where the string u is the period, $k > 1$ is an integer, and v is a proper prefix* (possibly empty) *of u. Let $|u| = p$ and suppose that all occurrence of the prefix $P[1, \cdots, 2p-1]$ in the text have already been found. Then all occurrences of the original pattern can be found using $O(n)$ additional operations and $O(1)$ additional time.*

Proof. The main substance of the computation below is searching for a pattern that is all ones.

Step 1. For each occurrence of $P[1, \cdots, 2p-1]$, at location i in the text, find whether it extends to an occurrence of $u^2 v$. If yes, mark bit $b_i := 1$ and otherwise mark $b_i := 0$.

Step 2. We partition the bits $b_1, \cdots, b_{n-2p-|v|+1}$ into p "strips." Strip s, $1 \leqq s \leqq p$, includes all bits whose index is $s \pmod p$ (for instance, strip 1 includes $b_1, b_{p+1}, b_{2p+1}, \cdots$).

Consider some location i in the text. The full pattern occurs at i if and only if the pattern $u^2 v$ occurs at all locations $i, i+p, \cdots, i+(k-2)p$. So, to find all occurrences of the full pattern, we simply must find every location in every strip whose bit is one and each of its successive $k-2$ bits is one. Step 3 shows how to do this for each of the strips in $O(n/p)$ operations and $O(1)$ time. Consider a strip s of length $t = n/p$.

Step 3.1. Partition the strip into $t/(k-2)$ successive subvectors of $k-2$ bits each.

Step 3.2. For each subvector find its largest and smallest zero bit in $O(k)$ operations and $O(1)$ time on a priority CRCW PRAM (which, as shown below, can be simulated on a common CRCW PRAM without asymptotic loss of efficiency).

Step 3.3. Given any bit b in the strip, the information (computed in Step 3.2 above) regarding the subvector containing the bit, as well as its successive subvector, suffices to determine in $O(1)$ operations whether all the $k-2$ successive bits of b are one.

Complexity. We have shown that any of our text analysis algorithms can be extended for the periodic case within additional $O(n)$ operations and $O(1)$ time on a common CRCW PRAM. Lemma 2.1 follows.

Common CRCW PRAM is enough. We describe our algorithms for the priority CRCW PRAM. In all kinds of instances but one, we can trivially use the common CRCW PRAM instead. Next, we characterize the one nontrivial kind of instances and show how to overcome the problem.

Consider the following problem. *Input.* Vector A of n bits. *Question.* Find the leftmost bit in A that is zero. Following [FRW88], we give an algorithm for this problem that needs $O(1)$ time and n processors on a common CRCW PRAM. (a) Partition A into $\sqrt{n}$ subvectors of length $\Theta(\sqrt{n})$ each. Using $O(n)$ operations find whether each of the subvectors has a zero bit. (b) Using $O(n)$ operations and $O(1)$ time, find the leftmost subvector containing a zero bit. For this, apply the parallel algorithm of [SV81] for finding the maximum among $m(=\sqrt{n})$ elements using m^2 processors in $O(1)$ time. (c) Apply the same algorithm for finding the leftmost zero bit in the leftmost subvector.

DEFINITION OF log* n. We denote the function symbol log by $\log^{(1)}$ and $\log^{(i)}$ is defined inductively as $\log \log^{(i-1)}$. Given a real number $r>1$, we define log* r to be the smallest integer i such that $\log^{(i)} r \leqq 2$. It is well known that the function log* is extremely slow in increasing and, for instance, $\log^* 2^{64000} = 5$.

DEFINITION OF THE PREFIX-SUMS PROBLEM [LF-80]. *Input.* Array of n numbers $[a_1, a_2, \cdots, a_n]$. *Problem.* Find all prefix-sums $a_1 + \cdots + a_i$, $1 \leqq i \leqq n$. Our parallel implementation applies parallel prefix-sums routines for the following problem. *Input.* Array of n numbers $[a_1, a_2, \cdots, a_n]$, where some of the n numbers are "marked" and the others are "unmarked." The problem is to compact all marked numbers into a shorter array. A standard technique in parallel computation (that was used in §§ 3 and 4 in [CV86], for instance) reduces this array compaction problem into the prefix-sums problem. The input for the prefix-sums problem is an array of n bits, where the value one represents a marked number and the value zero an unmarked number.

3. Pattern analysis. All algorithms in the present paper use the same *pattern analysis* stage.

Step 1. Find whether the pattern is periodic, and if yes find the period. Also compute array WITNESS.

Remark. For convenience, we assume that the pattern is nonperiodic throughout this presentation of the pattern analysis. However, if the pattern is periodic (let p be the length of the period), then by the nonperiodic prefix fact of § 2, the prefix of the pattern of length $2p-1$ must be nonperiodic. The computation of array WITNESS above, as well as the rest of the pattern analysis treats this prefix of length $2p-1$ as if it were the whole pattern.

Implementation and complexity. The pattern analysis of [Vi85] can be used for parallel computation of Step 1 in $O(\log m)$ time and $O(m)$ operations. Using the

pattern analysis from [BG88] the parallel time bound can even be improved to $O(\log \log m)$. We note that array WITNESS is needed only for the pattern analysis itself (Step 2 below) and can be deleted before proceeding to the text analysis.

The primary objective of the pattern analysis is the construction of a "*deterministic sample (denoted DS)*" of pattern positions. For presentation purposes we give full specification of the output of the pattern analysis only after Step 3.

We first define an auxiliary problem called *column sample.* This auxiliary problem helps us: (1) to define the deterministic sample; (2) to compute the deterministic sample. Step 2 finds a column sample. In Step 3 we derived the deterministic sample from the column sample.

Without loss of generality, suppose that m is even. Consider $m/2$ copies of the pattern $[c_1, c_2, \cdots, c_{m/2}]$ laid one on top of the other as in Fig. 3.1. The first location within copy c_2 is at the same *column* as the second location within copy c_1 and this correspondence extends to subsequent locations within c_1 and c_2. In general, the first location within copy c_i belongs to the same column as the ith location within c_1, and this correspondence extends to subsequent locations within c_1 and c_i.

Copy	$c_{m/2}$	:					–	⋯					–
	•	:											
	•	:											
	•	:											
Copy	c_3	:			–	⋯	–	⋯	–	–	–		
Copy	c_2	:		–	–	⋯	–	⋯	–	–			
Copy	c_1	:	–	–	–	⋯	–	⋯	–				
Column	#	:	1	2	3	⋯	$m/2$	⋯	m	$m+1$	$m+2$	⋯	$m+m/2-1$

FIG. 3.1

The *column sample* problem: select at most $\log m - 1$ columns $\overline{ds}(1), \cdots, \overline{ds}(l)$, $(l < \log m)$, and associate a character car $(\overline{ds}(i))$, with each column $\overline{ds}(i)$, $1 \leqq i \leqq l$, so that the following hold.

(1) There is exactly one copy c_j for which:
   (1.1) c_j intersects all these l columns (formally, $j \leqq \overline{ds}(i) < j+m$, for every $1 \leqq i \leqq l$).
   (1.2) for each column $\overline{ds}(i)$, the character in c_j equals the character associated with the column (formally, $P(\overline{ds}(i) - j + 1) = \text{car}(\overline{ds}(i))$, for every $1 \leqq i \leqq l$).

(2) For each of the other copies there is at least one column that intersects the copy and the character in the copy differs from the character associated with the column. (Formally, for each copy $c_k \neq c_j$, there is a column $\overline{ds}(i)$, $1 \leqq i \leqq l$, such that $k \leqq \overline{ds}(i) < k+m$ and $P(\overline{ds}(i) - k + 1) \neq \text{car}(\overline{ds}(i))$).

Example. See Fig. 3.2. A nonperiodic binary pattern of length $m = 16$ is given. The suggested column sample consists of column 11 with character 1, column 12 with character 0 and column 18 with character 1. The only copy that matches these three characters (at these columns) is the seventh copy marked as c_x.

Comments. (1) Step 2 shows the existence of a solution to the column sample problem. (2) The notation $\overline{ds}$ is used for the following reason. *ds* emphasizes the strong relation that exists between the column sample and the deterministic sample—our target problem. $\overline{ds}$ suggests that the column sample is not quite the deterministic sample.

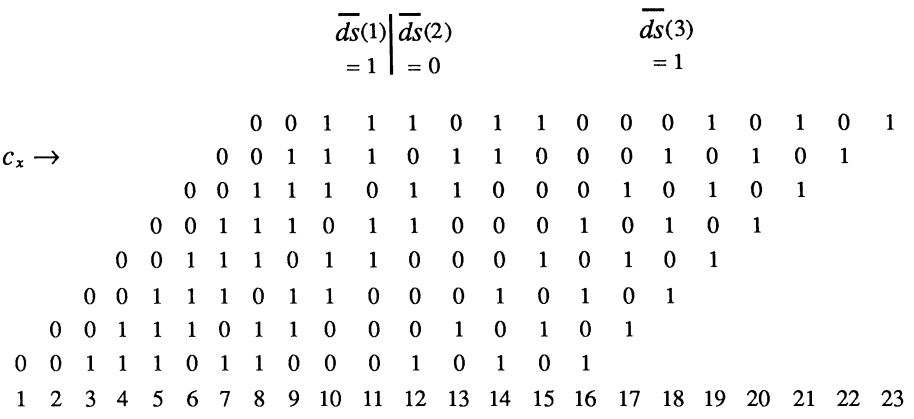

FIG. 3.2

The deterministic sample. Denote the copy that satisfies property (1) in the column sample problem by c_x. The *deterministic sample* is simply the column sample with respect to c_x. Formally, this sample is an ordered (not necessarily sorted) set $DS = [ds(1), ds(2), \cdots, ds(l)]$, of integers where $l \leqq \log m - 1$, and for each $1 \leqq j \leqq l$, $ds(j)$ is $\overline{ds}(j) - x + 1$.

Step 2. Step 2 inductively constructs sets $A_1, A_2, \cdots, A_l$, so that: (1) the base of the induction is the set $A_0 = \{c_1, c_2, \cdots, c_{m/2}\}$; (2) for each $0 \leqq i < l$, the set A_{i+1} is a nonempty subset of A_i and $|A_{i+1}| \leqq |A_i|/2$; (3) $|A_l| = 1$.

Inductive step. If A_i contains exactly one element then we set l, the cardinality of the column sample, to be i and proceed to Step 3. Otherwise, we build a nonempty subset A_{i+1} that contains at most one half of the elements in A_i. Let c_{leftmost} and $c_{\text{rightmost}}$ be the leftmost and rightmost copies in A_i, respectively. Array WITNESS will provide column $\overline{ds}(i+1)$ that contains two different characters in copies c_{leftmost} and $c_{\text{rightmost}}$. (Note that column $\overline{ds}(i+1)$ intersects every copy in A_i.) For each of these two characters find for how many copies in A_i the character in column $\overline{ds}(i+1)$ is equal to the character. Between these two characters, associate with column $\overline{ds}(i+1)$ the one with which less characters are equal. (Note that at most $|A_i|/2$ of the $|A_i|$ characters of the column will be equal to this character.) Set A_{i+1} is the subset of A_i containing all copies whose character at column $\overline{ds}(i+1)$ is equal to the selected character.

Example. Consider the construction of A_1. Note that c_1 and $c_{m/2}$, respectively, play the role of c_{leftmost} and $c_{\text{rightmost}}$, respectively. $\overline{ds}(1)$ is selected using WITNESS $(m/2)$.

Implementation and complexity. Suppose inductively that the copies belonging to A_i arrive in a compacted array (i.e., they have been renumbered from 1 to $|A_i|$). We use parallel prefix-sums for two purposes: (1) for each of the two characters of copies c_{leftmost} and $c_{\text{rightmost}}$ in column $\overline{ds}(i+1)$, finding the number of equal characters in the column (within the set A_i); and (2) to further compact the copies of A_{i+1} into an array of size $|A_{i+1}|$. Round $i+1$ takes $O(\log |A_i| / \log \log |A_i|)$ time and $O(|A_i|)$ operations using the prefix-sums algorithm of [CV89]. Since $|A_i|$ decreases geometrically, Step 2 takes a total of $O(\log^2 m / \log \log m)$ parallel time and $O(m)$ operations.

Step 3 (Deriving DS). Let c_x be the (only) element of A_l. The cardinality of DS is l and $DS = [ds(1), \cdots, ds(l)] := [\overline{ds}(1) - x + 1, \cdots, \overline{ds}(l) - x + 1]$.

Our text analyses will need the following information that was computed during the pattern analysis.

Output of the pattern analysis.

(1) The deterministic sample $DS = [ds(1), \cdots, ds(l)]$.

(2) Each set A_i, $1 \leqq i \leqq l$, in a compacted form. As indicated above, this means that the copies of A_i need to be renumbered from 1 to $|A_i|$.

Remark. Our optimal parallel algorithms use the sets A_i. The fact that the series $|A_0|$, $|A_1| \cdots |A_l|$ decreases geometrically is important for the efficiency of these algorithms.

Complexity of the pattern analysis. Since Step 2 dominates the complexity of the pattern analysis, we conclude that it needs $O(\log^2 m/\log\log m)$ time and $O(m)$ operations. The pattern analysis is given for the common CRCW PRAM. This model is used in Step 1 and in the prefix-sums computation of Step 2. A serial implementation needs $O(m)$ time.

Some practical considerations. The most important step for practical applications of the text analysis algorithms that follow is the actual set *DS* that is being constructed in Step 2. Particularly, we focus on the rate of reduction in the series $|A_i|$. In practice, it is most likely that the series $|A_i|$ may decrease much faster than by a factor of two. We mention in a nutshell a few common sense considerations in bringing this about. For instance, if the alphabet is of size $\sigma > 2$, we can have $|A_1|/|A_0| \leqq 1/\sigma$ by letting a character, that occurs in the pattern at most m/σ times, to guide us in the selection of a column. In general, it might be reasonable to invest more time in the pattern analysis and get a *DS* set that will facilitate a more efficient text analysis. For this, we may want to check all columns relative to each possible character. In each round of Step 2, we may even consider doing some backtracking (exhaustive search), where such investment makes sense. Curiously, in quite a few string matching algorithms (e.g., [Ga85a] or [W73]) the case where the alphabet is small is considered easier. The above considerations suggests that for our algorithms the opposite is correct.

Remarks. (1) Alon [A89] has constructed an example where the column sample problem needs $\Omega(\log m)$ columns. It is a nonperiodic binary sequence that is the output of a maximal linear feedback shift register.

(2) The proof of Theorem 1 in [A78] is remotely related to our deterministic sample construction. In principle, Adleman deals with a binary matrix. Looking for a small sample of columns he wants to rule out a match between a row of all zeros and any row listed in the input matrix. It is important to add that in his setting each row of the matrix is mostly ones. The crucial difference is that in our setting one of the input rows plays the role of the all zero row, and the computation needs to find such a row, since it is not known in advance which row will play this role. This row is chosen as the last survivor in the elimination process of rows according to residual minorities in columns. This explains why we feel that the deterministic sampling idea is new and only remotely related to Adleman construction.

3.1. Randomized pattern analysis. This section is not needed for understanding of the following sections. We suggest to skip it in a first reading of this manuscript.

We show how to perform the pattern analysis in $O(\log m)$ time and $O(m)$ operations, with high probability, by a randomized algorithm. The result will be a deterministic sample of size $O(\log m)$. A later comment explains why this can be guaranteed deterministically, and not only with high probability, and why all our text analysis results carry through. (Since the sample is drawn randomly, it would have been less confusing in this context to call it a *fixed*, rather than deterministic, sample.)

All our modifications refer to Step 2 above. We start Step 2, as before. We proceed, however, only until the size of the set A_i, of pattern copies, becomes at most $m/\log^2 m$. This requires $\gamma = O(\log\log m)$ rounds of the algorithm for the column sample problem.

Now, we switch to a randomized part. We outline modifications to the inductive step of Step 2. Our goal is similar. We construct smaller and smaller sets A_i. With high probability, the size of set A_{i+1} will be a constant fraction of the size of A_i. However, the difference is that we avoid performing prefix-sums, and therefore do not have compressed arrays or (deterministic) knowledge of their number of elements.

Remark. Avoiding prefix-sums computation is critical since prefix-sums need $\Omega(\log n/\log\log n)$ time using a polynomial number of processors. This was shown in [H86] together with the simulation result of [SV84], or directly in [BH87].

Finding c_{leftmost} and $c_{\text{rightmost}}$, the leftmost and rightmost copies in A_i, can be done in $O(1)$ time using $O(m/\log^2 m)$ operations on a priority CRCW PRAM. (Recall also the trick of [FRW88], as sketched in § 2, for simulation on a common CRCW PRAM.) Array WITNESS will provide the column $\overline{ds}(i+1)$, as before. The number of operations so far for each round is $O(m/\log^2 m)$ since we assign processors to jobs through the copies in A_γ.

In each round, the main effort is for selecting between two characters on column $\overline{ds}(i+1)$: either the character at copy c_{leftmost}, or the character at copy $c_{\text{rightmost}}$. We wish to select the character that is guaranteed to eliminate a constant fraction among the copies belonging to A_i with high probability. This is done in $O(1)$ time and $O(m/\log m)$ operations. The technique uses an idea from [Se89].

Overview. Let x_1 (respectively, x_2) be the number of copies in A_i whose character at column $\overline{ds}(i+1)$ is the same as at copy c_{leftmost} (respectively, $c_{\text{rightmost}}$). Note that the values of $x_1$and x_2 are unknown to us and we cannot compute them if we wish to implement each round in $O(1)$ time. Let $B[1,\cdots,(\log m)/2]$, be a vector of length $(\log m)/2$. For each integer j, $1 \leqq j \leqq (\log m)/2$, we assign the value zero to $B(j)$ with probability $x_1/(x_1+x_2)$, and the value one with probability $x_2/(x_1+x_2)$. This is done independently for different values of j, $1 \leqq j \leqq (\log m)/2$. We select for column $\overline{ds}(i+1)$ the character at copy c_{leftmost} if the total number of zeros in B is less than the total number of ones, and otherwise select the character at copy $c_{\text{rightmost}}$. This completes the overview. However, we still need to clarify several things.

(1) How to determine in $O(1)$ time whether the majority of the values in B are zero or one? For each of the $2^{(\log m)/2}$ possible binary vectors of length $(\log m)/2$, we precompute into a table the majority of zeros or ones using a total of $o(m)$ operations and $O(\log\log m)$ time. The size of the table is $2^{\log m/2}$ (which is $o(m)$). Using $(\log m)/2$ operations and $O(1)$ time per entry of the table (which is a binary vector of length $(\log m)/2$) we determine in each round whether the entry is identical with binary vector B. Determining whether vector B has more ones than zeros is done by table look-up.

(2) How to get the required probability for assignment of zero or one values to a random variable $B(j)$, $1 \leqq j \leqq (\log m)/2$? Given a random permutation of the elements in A_δ we assign processors to these elements through this permutation. Each processor standing by a copy of A_i whose character at column $\overline{ds}(i+1)$ is the same as the character at copy c_{leftmost} (respectively, $c_{\text{rightmost}}$) will try to write zero (respectively, one) at a variable $C(j)$. Since the priority CRCW PRAM is used we achieve the desired probability. Note that we will need a total of $O(\log^2 m)$ random permutations of the elements in A_δ for all rounds.

Comment. If we do not get the random permutations for free we can do the following. In [RGG89] it is shown how to generate a random permutation of n numbers in $O(\log n)$ time using $O(n\log n)$ operations on a CREW PRAM. So, had we taken

A_δ to be a set of at most $m/\log^3 m$ elements (instead of $m/\log^2 m$), we could have generated before the algorithm starts $O(\log^2 m)$ random permutations of $m/\log^3 m$ elements using a total of $O(m)$ operations and $O(\log m)$ time and have all other steps of this randomized pattern analysis carry through within the same efficiency bounds.

(3) Why our selection of the character for column $\overline{ds}(i+1)$ eliminates a constant fraction among the copies of A_i with high probability? For this we use a variant of Chernoff's bounds due to [AV79]. Each of the $y=(\log m)/2$ entries of vector B is an independent Bernoulli trial with probability of $p=x_1/(x_1+x_2)$ to get zero and $1-p$ to get one. We need only analyze cases where either p or $1-p$ are smaller than a fraction, say $f=1/10$ (since otherwise each of the two selections of a character for column $\overline{ds}(i+1)$ eliminates a fraction of at least f copies in A_i). Suppose $p<f$. We analyze the probability for getting a majority of zeros in vector B. Chernoff's bounds imply that the probability of getting at least $(1+\gamma)yp$ zeros in B is at most $\exp(-\gamma^2 yp/3)$ (exponent of the natural logarithm). We are interested in the case $\gamma=4$ and let us replace p by $f=1/10$, which is not smaller. The upper bound on the probability for getting a majority of zeros will be

$$\exp\left(-16\frac{\log m}{2}\frac{1}{10}\frac{1}{3}\right)\leqq\exp\left(-\frac{\log m}{4}\right)\leqq 2^{-(\log m)/3}=\frac{1}{m^{1/3}}.$$

With similar high probability, this process takes $O(\log m)$ rounds. Each round needs $O(1)$ time and $O(m/\log m)$ operations, totaling $O(\log m)$ time and $O(m)$ operations.

Observe that we are not yet done, since the text analysis needs to get each set A_i compressed into an array. This is achieved by means of performing a prefix-sums computation for each A_i that was obtained in the randomized part (i.e., $i>\gamma$). The main difference with respect to the deterministic Step 2 is that all these prefix-sums computations are performed *in parallel* after the entire deterministic sample and the series of A_i sets were computed. With high probability, we will have $O(\log m)$ parallel prefix-sums computations, performed in parallel. Each such computation needs $O(m/\log^2 m)$ operations and $O(\log m/\log\log m)$ time (since the assignment of processors to jobs is still through copies of A_γ). The total for the prefix-sums is $O(m/\log m)$ operations and $O(\log m/\log\log m)$ time with high probability.

Complexity. $O(\log m)$ time and $O(m)$ operations with high probability.

Comments. (1) The above algorithm is randomized. With high probability it runs in $O(\log m)$ time and $O(m)$ operations. But, what if we failed and got a sample in which $|A_{i+1}|\geqq(1-f)|A_i|$ for some i? (where f is the constant fraction that is guaranteed with high probability above.) In case this unlikely event happens, we add the following step to our randomized algorithm: run the deterministic pattern analysis. The time and number of operations bounds will remain the same, with high probability (because of the low probability of needing this additional step). An alternative to this additional step would be: repeat the randomized part until a "failure free" sample is derived.

So, obtaining a "good" sample is now guaranteed deterministically. Therefore, all our deterministic text analysis results will carry through.

(2) Yossi Matias suggested an alternative idea. Select to associate with column $\overline{ds}(i+1)$ between the character at copy c_{leftmost} and the character at copy $c_{\text{rightmost}}$ by simple coin tossing. At least one of these choices is guaranteed to eliminate one half of the copies in A_i. We can bound the probability of, say $\log m$, failures (a failure is when less than half are eliminated) in a sequence of $2\log m$ attempts by Chernoff bounds. However, what complicates (but does not make infeasible) adapting this simple

idea to our algorithms is that the cardinality of the sets A_i cannot be guaranteed to decrease geometrically at *each* round separately, with high probability.

4. Basic text analyses. We give two basic algorithms for analyzing the text: a *basic constant-time* algorithm and a *basic optimal speedup* algorithm. As implied by their names, these algorithms represent two "pure" extremes. The constant-time algorithm minimizes parallel time. It needs $O(1)$ time and $O(n \log m)$ operations. The optimal speedup algorithm minimizes the total number of operations. It needs $O(\log m)$ time and $O(n)$ operations. The next section shows how to combine ideas from both algorithms for getting $O(\log^* n)$ time and $O(n)$ operations. Unless otherwise stated, we assume that the pattern is nonperiodic. Section 2 explains how to extend any of our alternative text analyses to the periodic case. For both algorithms below we partition the first $n-m+1$ locations of the text into successive substrings of exactly $m/2$ positions each (and perhaps one substring of fewer positions). Initially, any position in the text is a *candidate* for being the start of an occurrence of the pattern. We will assume throughout that $n \geqq 3m/2$ (so that there is at least one $m/2$ block of initial candidates).

Basic constant-time text analysis.

Step 1. For each position $1 \leqq i \leqq n-m+1$ in the text, check whether the following $l = |DS|$ equalities hold: $T(i-1+ds(j)) = P(ds(j))$ for every $ds(j) \in DS$. If any of these equalities does not hold, we eliminate location i as candidate.

Step 1 needs at most $\log m$ checks per any of the $n-(m-1)$ candidates, or a total of $O(n \log m)$ checks.

Next, we make a detour and present a key property of the deterministic sample. Let x be the same as in Step 3 of the pattern analysis (i.e., copy c_x is the only shift of the pattern that matches the column sample).

The Ricochet property. Let i be a candidate location in the text following Step 1. Then, based only on the candidacy of location i, we can eliminate any remaining candidate in the $x-1$ locations preceding i, as well as the $m/2-x$ locations succeeding i (that is, location $i-x+1$ through $i-1$ and $i+1$ through $i+m/2-x$).

The word "ricochet" is meant to convey the following. A candidate location is determined using matches with the deterministic sample (that consists of at most $\log m$ locations). Still this direct match of at most $\log m$ locations allows for indirect "ricochet-like" hit (or elimination of candidacy) of many (up to $m/2$ in number) locations.

Step 2. For each successive substring of length $m/2$, find its leftmost and rightmost candidates on a priority CRCW PRAM (which, in turn, can be simulated on a common CRCW PRAM without asymptotic loss of efficiency). Based on the Ricochet property, disqualify all candidates that are neither leftmost nor rightmost in their substring.

Step 2 results in having at most two candidates per any successive substring of size $m/2$. So finally, we have Step 3.

Step 3. Apply a character-by-character check to each candidate location.

Complexity of the basic constant time text analysis. $O(1)$ time using $n \log m$ processors on the common CRCW PRAM.

Basic optimal speedup (and linear serial) text analysis.

Outline. Our goal is to reduce the total number of operations from $O(n \log m)$ to $O(n)$. A first attempt at this problem is to perform Step 1 of the basic constant-time algorithm in $l = |DS|$ rounds, as follows. The input for round α is all text positions (candidates) that matched the first $\alpha - 1$ positions of the deterministic sample (i.e.,

positions $ds(1), \cdots, ds(\alpha-1)$ of the pattern). In round α, check each candidate against the αth position, $ds(\alpha)$, of the deterministic sample. Unfortunately, this attempt does not lead to a bound smaller than $O(n \log m)$ on the number of operations. We overcome this problem, as follows. Each round will also include "Ricochetlike" diqualification of candidates, in the spirit of Step 2 above. This will lead to $O(n/2^{\alpha})$ candidates following round α, hence a total of $O(n)$ operations.

Step 1. For each position $1 \leqq i \leqq n-m+1$, in the text, check whether the following equality holds: $T(i-1+ds(1)) = P(ds(1))$.

In Step 2 below, we focus on a single (successive) substring of length $m/2$. All such substrings are treated similarly, and simultaneously in parallel.

Step 2.1. Find the leftmost candidate a, and rightmost candidate b in the substring. (Formally, a is the smallest index in the substring for which $T(a-1+ds(1)) = P(ds(1))$, and b is the largest such index.) Location $lg = a-1+ds(1)$ in the text is called a *left guide* and location $rg = b-1+ds(1)$ is called a *right guide.*

Consider the set of pattern positions (or shifts) $A_1 = [c_{1,1}, c_{1,2}, \cdots, c_{1,|A_1|}]$ that was obtained in the pattern analysis. We will construct two sets of text positions T_{lg} and T_{rg}. Guiding location lg induces set T_{lg} using set A_1. T_{lg} will simply be the set of $|A_1|$ text locations that align under positions of set A_1, when we align location $\overline{ds}(1)$ (this is the first location in the column sample of the pattern analysis) at the same column as lg in T. Fig. 4.1 illustrates four things: (1) the diamond-shaped structure at the top is similar to Fig. 3.1; (2) column $\overline{ds}(1)$ in the column sample and location lg in T align at the same column; (3) members of set A_1 in the pattern align at the same columns as members of the set T_{lg}; (4) location a in T is a member of T_{lg}. (Location a in T and column $c_{1,z}$ must align at the same column, for some $c_{1,z}$ that is a member of set A_1.)

Similarly, guiding location rg induces set T_{rg} using set A_1. T_{rg} is the set of text locations that are aligned with set A_1 when location $\overline{ds}(1)$ in the diamond shape is aligned at the same column as location rg in T.

The key correctness observation. Consider a location in the substring of the text. If it is neither in set T_{lg} nor in set T_{rg}, then it cannot be a start of an occurrence.

Proof. The observation follows from the following facts: (1) $b-a < m/2$. (2) Occurrence of the pattern can be in one of the $x-1$ text locations preceding b, only if the text location is in T_{rg}. (3) Occurrence can be in one of the $m/2-x$ text locations succeeding a, only if the text location is in T_{lg}. (4) There is no occurrence in locations of the substring that precede a or succeed b (by the selection of a and b).

Throughout the algorithm we mark as noncandidates locations of the text for which our computation indicates that occurrence is impossible (e.g., in Step 1 above). A possibly confusing fact is that sets T_{lg} and T_{rg} themselves may include locations that are already noncandidates. To straighten out our terminology we refer to text locations in the T_{lg} and T_{rg} lists that are noncandidates as *straw* candidates.

Step 2.2. Using set A_1 construct the set of text positions T_{lg} (respectively, T_{rg}) that can co-exist with selecting the first entry of the column sample $\overline{ds}(1)$ aligned at the same column as location lg in T (respectively, location rg in T).

Implementation remark. Assignment of processors to jobs is always a concern in designing parallel algorithms. This concern is even more acute for algorithms whose target running time prohibit application of prefix-sums, as here: we implement each round below in constant time while prefix-sums need $\Omega(\log n/\log\log n)$ time using a

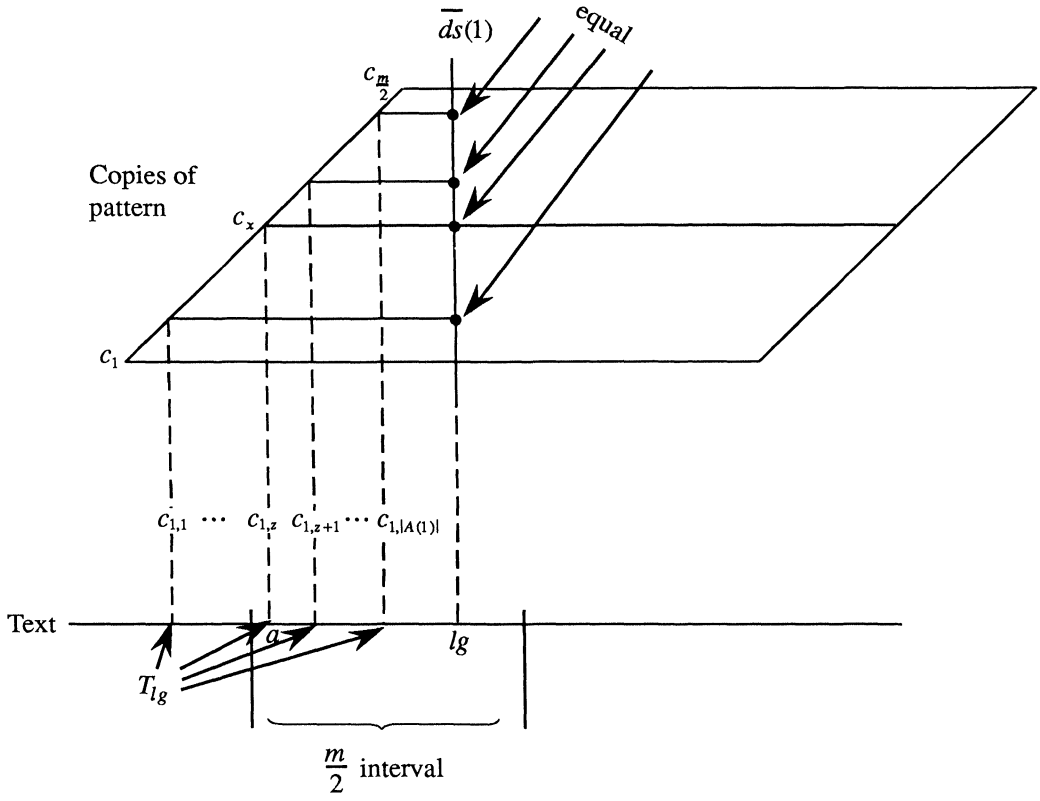

FIG. 4.1

polynomial number of processors (see references in an earlier remark). A later comment explains why it led us to include straw candidates in the T_{lg} and T_{rg} lists.

Following the above first round, Steps 1 and 2 are iterated in $l-1$ more rounds. In each round below, we focus on a single substring of length $m/2$. Other substrings are treated similarly, simultaneously in parallel. Here is an outline of round α, for $2 \leqq \alpha \leqq l$.

Step 1′. Consider every nonstraw candidate i in list T_{lg} or T_{rg} of round $\alpha-1$. For each such candidate, check whether the following equality holds: $T(i-1+ds(\sigma)) = P(ds(\alpha))$.

Step 2′.1. Find the leftmost remaining candidate, a in T, and rightmost remaining candidate, b in T, in the substring. (Formally, a is the smallest index in the substring of a candidate for which $T(a-1+ds(\alpha)) = P(ds(\alpha))$, and b is the largest such index.) Location $lg = a-1+ds(\alpha)$ in the text is called a *left guide* and location $rg = b-1+ds(\alpha)$ is called a *right guide.*

Step 2′.2. Using set A_α construct the set of text positions T_{lg} (respectively, T_{rg}) that can co-exist with selecting the αth entry of the column sample, that is column $\overline{ds}(\alpha)$, aligned at the same column as location lg in T (respectively, rg in T).

Complexity of Step 2. Since $m/2^{\alpha+1}$ is a bound on the number of elements in each T_{lg} or T_{rg} list, the bound on the total number of operations decreases by a factor of

at least two in each round. Therefore, the total number of operations is $O(n)$ and the time is $O(\log m)$. The only nontrivial detail in an exact parallel implementation of this algorithm is the issue of assignment of processors to elements of any T_{lg} and T_{rg} list in any round α.

Assignment of processors. We assign processors to the element of the T_{lg} and T_{rg} lists through the indices of the set A_α. (That is, we get every index of T_{lg} by means of adding lg to every index of A_α.) The trick is that these indices were computed in the pattern analysis (using prefix-sums). Observe that a processor will also be assigned to each straw candidate of each T_{lg} and T_{rg} list in each round. Such processor simply remains idle during the round.

The following comments shed some more light on the rounds of Step 2.

Comment 1. The key correctness observation holds also following each Step 2'.1 of each round. Specifically, each candidate at the substring of length $m/2$ must lie either in T_{lg} or T_{rg}.

Comment 2. Again, the T_{lg} and T_{rg} lists may include straw candidates. Straw candidates may come from three sources: (i) They were not in the T_{lg} or T_{rg} list for any guiding location of round $\alpha-1$. (ii) They were already straw candidates in the T_{lg} or T_{rg} list for some guiding location of round $\alpha-1$. (iii) They were candidates in the T_{lg} or T_{rg} list for some guiding location of round $\alpha-1$ but they failed the check of Step 1' in round α.

At this stage, we remain with at most $2\lceil (n-m+1)/(m/2)\rceil = O(n/m)$ candidates.

Step 3. Compare the whole pattern relative to each candidate, in a naive character-by-character manner.

Complexity of the basic optimal speedup text analysis. $O(\log m)$ time using an optimal number of processors on the common CRCW PRAM.

5. Optimal $O(\log^* n)$ time text analysis. We show how to perform the text analysis in $O(\log^* n)$ time and $O(n)$ operations. The algorithm will have three stages. The main part (Stages 1 and 2) applies the *accelerating cascades* design principle, as discussed in [CV86].

Stage 1. Run Steps 1 and 2 of the optimal speedup basic text analysis for $2\log\log^* n+2$ rounds. For this, we use the first $\delta := 2\log\log^* n+2$ positions of *DS*, the deterministic sample. The variable δ will keep track of the number of positions of *DS* that have already been "used" in Step 2 as well. The total number of elements (candidates and straw candidates) in the resulting T_{lg} and T_{rg} lists will be at most $n/(\log^* n)^2$.

Complexity. $O(n)$ operations and $O(\log\log^* n)$ (which is $o(\log^* n)$) time.

Stage 2 has $\log^* n$ iterations, each limited to constant time. The input for each iteration is a set of candidates (in T_{lg} and T_{rg} lists). As iterations proceed, the number of candidates decreases and we can apply an increasing number of tests, per each candidate at hand, in order to accelerate the candidate disqualification rate. Interestingly, while the overall serialization of events in Stages 1 and 2 together is motivated by the basic optimal speedup text analysis, each iteration of Stage 2 resembles Step 1 of the basic constant-time text analysis, where several positions from *DS* are checked at once. Stage 3 is the same as Step 3 in the basic optimal speedup text analysis.

Stage 2.

for *count* := $\log^* n$ *downto* 1 *do*

(*Input for present iteration*: Total of at most $n/(\log^{(\text{count})} n \log^* n)$ (straw and nonstraw) candidates in the T_{lg} and T_{rg} lists.)

For each (nonstraw) candidate i, check the next $\log^{(\text{count})} n$ positions in the sample $DS = [ds(1), \cdots, ds(l)]$.
Specifically, check whether the following $\min\{l-\delta, \log^{(\text{count})} n\}$ equalities hold:
$T(i-1+ds(\delta+1)) = P(ds(\delta+1))$,
$T(i-1+ds(\delta+2)) = P(ds(\delta+2))$,
$\cdots$
$T(i-1+ds(\min\{l, \delta+\log^{(\text{count})} n)) = P(ds(\min\{l, \delta+\log^{(\text{count})} n))$.
For each substring of length $m/2$, find its leftmost (nonstraw) candidate and its rightmost (nonstraw) candidate. Get from them the *guiding* locations and the lists T_{lg} and T_{rg}.

Since this procedure exhausts the sample DS, we end up with just one candidate in each T_{lg} and T_{rg} list.

Complexity. In iteration *count* we perform at most $O(\log^{(\text{count})} n)$ operations per each of the iteration's input candidates in $O(1)$ time. Since the number of such input candidates is bounded by $n/(\log^{(\text{count})} n \log^* n)$, we get $O(n/\log^* n)$ operations and $O(1)$ time per iteration, or a total of $O(n)$ operations and $O(\log^* n)$ time.

Stage 3. Compare the whole pattern relative to each candidate, in a naive character-by-character manner.

Comment. Actual computation of $\log^* n$. All functions used in this paper can be computed within the complexity bounds claimed here. We refer the reader to [BV89] that shows how to compute the function $\log^* n$, for instance, in constant time using n processors.

6. Further research and speculation.

(1) A possibly sublinear serial implementation. Rivest [Ri77] showed that, under some assumptions about the string matching algorithm, sublinear time cannot be achieved in the worst case, if the pattern is considerably shorter than the text. On the other hand, there were a few works whose concern was to show that some string matching algorithms need sublinear time under some assumptions about the source of the input. The difficulty about these works is that they make assumptions on what a typical input looks like. We did not find satisfactory ways for making assumptions of this kind. To demonstrate our difficulty, we show why the common probabilistic assumption that each character of the test is equally likely and that all positions are probabilistically independent does not make sense, in general. This assumption implies that if the length of the pattern is not very small relative to the length of the text, the probability of having an occurrence of the pattern is extremely small. However, in many string matching problems we have no doubt that occurrences exist and only need to find them!

Consider a serial implementation of the basic optimal speedup text analysis algorithm. We already mentioned that it runs in linear time. Still, we provide some practical ideas for enhancing its performance. Observe that if we find a match between a text character and a pattern character in Step 1 (or Step 1′) then we can immediately use it for reducing the number of candidates near this location of the text. This may save some additional comparisons between characters of the text and the pattern in the present round. In addition, recall the remark on practical considerations in § 3. The above discussion explains why, unfortunately, we do not see how to explore these ideas in a theoretically sound manner.

(2) Extension to two or higher dimensions. Suppose our pattern is a two-dimensional m-by-m array. We are not familiar with successful attempts to extend the

concept of periodicity in strings to higher dimensions. The following *conflicting occurrences assumption* resembles the case of nonperiodic patterns in strings. Consider any two positions in the text (i,j) and (i_1,j_1) that are close enough. Formally, we require that $|i-i_1| \leqq m/2$ and $|j-j_1| \leqq m/2$. Consider laying one copy of the pattern to start at (i,j) and another copy to start at (i_1,j_1). The intersection of these two copies contains (possibly several) arrays of size $m/2$ by $m/2$. The assumption about the pattern array concerns each of these $m/2$-by-$m/2$ arrays. The assumption is that the $m/2$-by-$m/2$ array must contain a position in which the two copies of the pattern have two different characters. (This resembles the information in WITNESS.) Such an assumption make it possible for our algorithmic approach to carry through efficiently.

(3) Extension to approximate matches. Consider again the two-dimensional case, where the pattern and text consist of arrays of pixels, where each pixel is characterized by its grayness (or intensity). Suppose that there are several levels of grayness. A natural concept of approximate, rather than exact, match is where only "very different" levels of grayness are defined to mismatch. (The problem is that a small difference between two levels of grayness is insufficient evidence for a mismatch.) A possible conflicting occurrences assumption will be similar to the above-suggested assumption for extensions to higher dimensions. Such an assumption would make it possible for our algorithmic approach to carry through efficiently.

(4) Speculations on complexity measures. Machine vision is one of the most frustrating application fields for any algorithm designer, for our performance as humans analyzing scenes is vastly superior to any algorithm presently imaginable for even the most powerful machines. Our algorithm may shed some light in attempting to explain this phenomenon. Power of computing machinery is often measured by number of arithmetic operations per second and other traditional computational intensity measures. Advances in computer architecture are geared to optimize such measurements and indeed computers greatly outperform humans for computationally intense tasks. On the other hand, human vision is supposedly very effective in a few very simple tasks, such as sampling a point of reference (e.g., "pick a red car in an aerial photo of a huge parking lot") and large fan-in AND (e.g., "are all cars in the parking lot red?") or OR.

We review our basic constant time text analysis. We show that it uses very degenerate computations and barely performs any "real" computation. Rather it can be implemented using only the simple tasks that humans seem to perform well. Step 1 compares characters and then takes the AND of $|DS|$ bits. (An even more "humanlike" approach would be to "associatively identify" the deterministic sample. By this we mean that given a small pattern it might be interesting to consider hypothetical computers that can retrieve, by means of a unit-cost operation, occurrences of the pattern.) Step 2 selects a leftmost (and rightmost) bit whose value is one out of each substring of $m/2$ bits. (Again, such leftmost bit can be associatively identified.) This already makes possible occurrence of the pattern very sparse. Finally, Step 3 takes the AND of m bits to verify occurrences.

This may suggest that for pattern recognition tasks, it might be less appropriate to restrict attention to conservative computational intensity measures only, but rather articulate new measures as yardsticks for novel and potent computer architectures.

Acknowledgments. Helpful discussions with Noga Alon, Amihood Amir, Gary Benson, Omer Berkman, Joseph Ja'Ja', Rao Kosaraju, Gadi Landau, Yossi Matias, Azriel Rosenfeld, and Ramakrishna Thurimella are gratefully acknowledged.

REFERENCES

[A78] L. ADLEMAN, *Two theorems on random polynomial time*, in Proc. 19th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1978, pp. 75–83.

[A89] N. ALON, personal communication.

[AV79] D. ANGLUIN AND L. G. VALIANT, *Fast probabilistic algorithms for Hamiltonian circuits and matching*, J. Comput. Systems Sci., 18 (1979), pp. 155–193.

[BG88] D. BRESLAUER AND Z. GALIL, *An optimal $O(\log \log n)$ time parallel string matching algorithm*, preprint, 1988. Also appeared as a chapter in O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, *Highly-Parallelizable Problems*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 309–319.

[BH87] P. BEAME AND J. HASTAD, *Optimal bounds for decision problems on the* CRCW PRAM, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 83–93.

[BR89] B. BERGER AND J. ROMPEL, *Simulating* $(\log^2 n)$*-wise independence in* NC, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989, pp. 2–7.

[BV89] O. BERKMAN AND U. VISHKIN, *Recursive *-tree parallel data-structure*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989, pp. 196–203.

[BV90] ———, *On parallel integer merging*, Tech. Report UMIACS-90-15, Institute for Advanced Computer Studies, University of Maryland, College Park, MD, January 1990.

[BM77] R. S. BOYER AND J. S. MOORE, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 762–772.

[CV86] R. COLE AND U. VISHKIN, *Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 206–219.

[CV89] ———, *Faster optimal prefix sums and list ranking*, Inform. and Comput., 81 (1989), pp. 334–352.

[FRW88] F. E. FICH, R. L. RAGDE, AND A. WIGDERSON, *Relations between concurrent-write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[Ga85a] Z. GALIL, *Optimal parallel algorithms for string matching*, Inform. and Control, 67 (1985), pp. 144–157.

[Ga85b] ———, *Open problems in stringology*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Springer-Verlag, Berlin, New York, 1985, pp. 1–8.

[Gi59] A. GILL, *Minimum-scan pattern recognition*, IRE Trans. Inform. Theory, 5 (1959), pp. 52–58.

[GS83] Z. GALIL AND J. I. SEIFERAS, *Time-space-optimal string matching*, J. Comput. System Sci., 26 (1983), pp. 280–294.

[H86] J. HASTAD, *Almost optimal lower bounds for small depth circuits*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 6–20.

[KLP89] Z. M. KEDEM, G. M. LANDAU, AND K. V. PALEM, *Optimal parallel suffix-prefix matching algorithms and applications*, in Proc. 1st ACM Symposium on Parallel Algorithms and Architectures, Association for Computing Machinery, New York, 1989, pp. 388–398.

[KMP77] D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 322–350.

[KR87] R. M. KARP AND M. O. RABIN, *Efficient randomized pattern-matching algorithms*, IBM J. Res. Develop., 31 (1987), pp. 249–260.

[LF80] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computations*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[LS62] R. C. LYNDON AND M. P. SCHUTZENBERGER, *The equation $a^M = b^N c^P$ in a free group*, Michigan Math J., 9 (1962), pp. 289–298.

[Lu88] M. LUBY, *Removing randomness in parallel computation without a processor penalty*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 162–173.

[MNN89] R. MOTWANI, J. NAOR, AND M. NAOR, *The probabilistic method yields deterministic parallel algorithms*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989, pp. 8–13.

[Ra76] M. O. RABIN, *Probabilistic algorithms*, in Algorithms and Complexity, J. F. Traub, ed., Academic Press, New York, 1976, pp. 21–39.

[RGG89] V. RAJAN, R. K. GHOSH, AND P. GUPTA, *An efficient parallel algorithm for random sampling*, Inform. Process. Lett., 30 (1989), pp. 265–268.

[Ri77] R. L. RIVEST, *On the worst-case behavior of string-searching algorithms*, SIAM J. Comput., 6 (1977), pp. 669–674.

[Se89] S. SEN, *Finding an approximate-median with high-probability in constant time*, manuscript, 1989.

[St88] Q. STOUT, *Constant-time geometry on* PRAMs, in Proc. Internat. Conference on Parallel Processing, Chicago, IL, 1988.

[SV81] Y. SHILOACH AND U. VISHKIN, *Finding the maximum, merging and sorting in a parallel model of computation*, J. Algorithms, 2 (1981), pp. 88–102.

[SV84] L. J. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 409–422.

[U85] E. UKKONEN, *Finding approximate patterns in strings*, J. Algorithms, 6 (1985), pp. 132–137.

[Va75] L. G. VALIANT, *Parallelism in comparisons models*, SIAM J. Comput., 4 (1975), pp. 348–355.

[Vi85] U. VISHKIN, *Optimal parallel pattern matching in strings*, Inform. and Control, 67 (1985), pp. 91–113.

[W73] P. WEINER, *Linear pattern matching algorithm*, in Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, IEEE Computer Society, Washington, DC, 1973, pp. 1–11.