PARALLEL SUFFIX-PREFIX MATCHING ALGORITHM AND APPLICATIONS[*]

ZVI M. KEDEM [†], GAD M. LANDAU [‡], AND KRISHNA V. PALEM [§]

Abstract. Our main result in this paper is a parallel algorithm for *suffix-prefix matching* that has optimal speedup on a CRCW PRAM. Given a string of length m the algorithm runs in time $O(\log m)$ using $m/\log m$ processors. This algorithm is important because we utilize suffix-prefix matching as a fundamental building block to solve several pattern and string matching problems such as: *1. String Matching, 2. Multi-text/Multi-pattern string matching, 3. Multi-dimensional Pattern Matching, 4. Pattern Occurrence Detection, 5. On-line String Matching.* In particular, our techniques and algorithms are the first to preserve optimal speedup in the context of pattern matching in higher dimensions, and are the only known ones to do so for dimensions $d > 2$.

Key words. Amortized Complexity, CRCW PRAMs, Multidimensional Pattern Matching, Parallel Algorithms, Pattern-matching Automaton, Speedup, String Matching

AMS subject classifications. 68P99, 68Q10, 68Q22, 68Q25, 68Q68, 68R15, 68U15

[*] A preliminary version of this paper appeared in Proceedings of 1989 ACM Symposium on Parallel Algorithms and Architectures

[†] Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185; (212) 998-3101; kedem@cs.nyu.edu. The research of this author was supported in part by ONR under contract number N00014-85-K-0046 and by NSF under grant number CCR-89-6949

[‡] Department of Computer Science, Polytechnic University, 333 Jay St., Brooklyn, NY 11201; (718) 260-3154; landau@pucs2.poly.edu. The research of this author was partially supported by NSF grants CCR-8908286 and CCR-9305873 and by the New York State Science and Technology Foundation, Center for Advanced Technology in Telecommunications, Polytechnic University, Brooklyn, NY.

[§] Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185; (212) 998-3512; palem@cs.nyu.edu.

1. Introduction. Several important problems in computing involve the detection of repeated patterns within regular structures such as strings and higher dimensional arrays. As a consequence, there is a rich history of fast algorithms for solving these problems. Karp, Miller and Rosenberg [KMR72] used techniques based on successively refining equivalence classes of patterns of increasing size, where the patterns in an equivalence class are identical. Initially, they consider equivalence classes of patterns made up of a single character from the input. On each successive step, they construct equivalence classes of bigger pieces of the input by appropriately combining the equivalence classes from the previous step. Their algorithm was not optimal. Weiner, using a different approach [W73], built a suffix tree and used it to design a linear time algorithm for a fixed alphabet.

Two linear time (optimal) algorithms for the *string matching* problem were given by Knuth, Morris and Pratt [KMP77] and Boyer and Moore [BM77] . These algorithms are based on the powerful notion of a *failure function*. Failure functions led to a substantial amount of subsequent work. Galil and Seiferas [GS83] have reported time and space optimal sequential, real time algorithms for string matching. Aho and Corasick [AC75] have linear time algorithms for a natural generalization of the string matching algorithm in which the input has multiple patterns, possibly of different sizes.

Efficient algorithms for *multi-dimensional pattern matching* (or *d-dimensional pattern matching*) have been independently reported by Baker [Ba78], Bird [Bi77], and Karp and Rabin [KR87] (randomized). These algorithms run in time[1] $O(d(n^d + m^d))$, given that the input text and pattern are respectively of size n^d and m^d.

Parallel algorithms for string matching were given by Galil [G84] for strings from a fixed alphabet, and Vishkin [V85] for strings from an arbitrary alphabet. Later Breslauer and Galil [BG90], Vishkin [V91], and Galil [G92] designed new parallel algorithms for string matching with an arbitrary alphabet. Mathies [M88], and Amir and Landau [AL88] have presented parallel algorithms for solving the multi-dimensional pattern matching problem. However, the techniques developed in these algorithms do not scale in the sense that they have not yielded optimal speedup when applied to pattern matching in higher dimensions. Our results described in this paper were the first to solve pattern matching problems in higher dimensions, with optimal speedup. Subsequently and recently Amir et. al, [ABF93] and Cole et. al, [CCG+93] presented techniques for achieving optimal speedup for the two-dimensional pattern matching problem with an unbounded alphabet.

Our main result is an optimal speed-up parallel algorithm for solving the *suffix-prefix matching problem*. Using this algorithm as the basic building block, we specify optimal speedup parallel algorithms for several pattern and string matching problems; optimal speedup parallel algorithms were not known for most of these problems before. (Given a problem instance of size n, we say that a parallel algorithm running in $T(n)$ steps using $P(n)$ processors performs *work* $P(n) \times T(n)$. We say that such an algorithm has *optimal speedup* if the work that it performs is (asymptotically) the same as the *running time* of the best-known *sequential* algorithm for solving the same problem.) The definition of the suffix-prefix matching problem and a list of these other problems are given in Section 1.1. Our results have the advantage that they scale to higher dimensions while preserving optimal speedup.

In particular, our algorithm relies on the appropriate combination of two basic ideas. First, a novel aspect of our algorithm is that we construct a finite automation as in Aho and Corasick [AC75], in parallel. This finite automaton can be used to recognize short strings of length $O(\log m)$, where m is the size of the input, in linear work. Since this step uses failure functions, we note that our algorithm is the first to use them in the parallel context. The second idea that we use involves computing *characteristics*, originally introduced in [AILSV88] and [KP92]. These characteristics are

[1] In their paper Karp and Rabin state that these ([Ba78][Bi77], and [KR87]) algorithms run in time $O((n^d + m^d))$, but Richard Karp, in personal communication, clarified that they assume a constant d.

essentially the "short names" used by Karp, Miller and Rosenberg [KMR72], as described in the first paragraph above. It is this combination that allows us to obtain optimal speedup parallel algorithms for pattern matching in higher dimensions. All our algorithms are developed in the context of an arbitrary CRCW PRAM[J92].

The rest of this paper is organized as follows. In Section 1.1 the significant results in this paper are reported. Section 1.2 describes some previous work. Section 2 presents the new optimal speedup algorithm for the s-p problem. Section 3 describes some applications of the s-p matching algorithm. Section 4 gives some concluding remarks.

1.1. Significant results in this paper.

Our contributions are:

1. We specify a parallel algorithm for the *suffix-prefix matching problem* (or simply the *s-p matching problem*) that has optimal speedup. The input to this problem consists of two equal length strings A and B, both of length m. We wish to determine for each i where $1 \leq i \leq m$, whether the suffix of A of size i is identical to the prefix of B of the same size. Our algorithm runs in $O(\log m)$ time using $m/\log m$ processors. The s-p matching problem embodies a computational bottleneck in several pattern matching problems. Therefore, using this algorithm as the fundamental building block, we are able to design the following optimal speedup parallel algorithms.
2. We specify a new and simple algorithm that has optimal speedup, for solving the string matching problem with a polynomial size alphabet. The input to this problem consists of two strings: a text of length n and a pattern of length m ($m \leq n$). We wish to determine for each position of the text, whether the pattern is equal to the substring of the text starting at it. Our algorithm runs in $O(\log m)$ time using $n/\log m$ processors. Except for this problem, optimal speedup parallel algorithms were not known for the remaining four applications listed below.
3. We specify a new and simple algorithm that has optimal speedup, for solving the *multi-text/multi-pattern string matching* problem. The input to this problem consists of u text strings $T_1, T_2, \ldots, T_u$, respectively of lengths $n_1, n_2, \ldots, n_u$, and v patterns $P_1, P_2, \ldots, P_v$, each of length $m \leq n_i$ for $1 \leq i \leq u$. We wish to determine for each position of each text string, whether one of the patterns is equal to the substring of the text starting at it. Our algorithm runs in time $O(\log m)$ using $(vm + \sum_{j=1}^{u} n_j)/\log m$ processors.
4. We specify an optimal speedup parallel algorithm for *multi-dimensional pattern matching*. The input to this problem consists of two arrays: a text of size n^d and a pattern of size m^d ($m \leq n, d \geq 1$). We wish to determine for each position of the text, whether the pattern is equal to the subarray, of size m^d, of the text starting at it. This algorithm runs in time $O(d \log m)$ using $n^d/\log m$ processors.
5. We specify an optimal speedup parallel algorithm for solving the *occurrence detection* problem. Informally, the input to this problem is a *text string* of size n, presented as an (unordered) set of l *pieces* of size $k = n/l$, no two of them equal. The problem is: given a pattern of size m, determine if there exists a concatenation of the l substrings to form a single string of size n, such that the pattern occurs in the resulting string. This question is relevant to issues in molecular biology [CD88], [TU88]. Our algorithm solves this problem in time $O(\log m)$ using $n/\log m$ processors.
6. We specify a parallel algorithm for *on-line string matching* that has optimal *amortized* speedup. This is an on-line algorithm motivated by practical problems such as text editing. Here, we have a text of size n and a pattern of size $m \leq n$ for which the string matching problem has already been solved. Now, the text is extended by k characters, i.e., it is now of length $n + k$, and we wish to determine if there are any additional matches of the pattern in the text. Our algorithm solves this problem in time $O(\log m + \log k)$ and the *amortized*

work (to be defined precisely later) done by it is always linear, and hence optimal.

1.1.1. Remarks on alphabet size and space:. Throughout the rest of this paper, we are concerned with an alphabet size that is at most polynomial in m. Given this, we assume without loss of generality that $\Sigma \subseteq \{0, 1, \ldots, m-1\}$ and therefore the individual symbols of the alphabet (elements of Σ) are each $\log m$ bits long. Following standard conventions[J92], a processor is assumed to be able to read a constant number of symbols in a constant number of time units. We note that this is more general than the constant sized alphabet used in [G84] for example. However, we do not consider alphabets of an arbitrary size relative to the size of the string, as in [V85]. All of our operations are standard PRAM operations. In contrast, in dealing with an arbitrary sized alphabet (as in [V85]), it is assumed that any two symbols can be tested for equality in $O(1)$ time and work, independent of the size. Given this assumption, we note that a problem with an arbitrary alphabet could be converted to one in which $|\Sigma| \leq m$ by sorting, in $O(n \log m)$ work.

All of our techniques will work with space $O(m^{1+\epsilon})$ as in the work of Apostolico et al. [AILSV88], for any $0 < \epsilon \leq 1$, with a corresponding slow-down proportional to $1/\epsilon$. These implementations with reduced space requirements can be realized with just $O(m)$ cost (work) for initialization, following [H88].

1.2. Previous work. We now compare our applications of the s-p matching algorithm with previous results.

1. Galil [G84] and Vishkin [V85] have designed optimal speedup parallel algorithms for string matching, with input strings drawn respectively from a bounded and arbitrary alphabet. Both of these algorithms run in time $O(\log n)$ using $n/\log n$ processors of a CRCW PRAM. Breslauer and Galil [BG90], designed an algorithm that runs in time $O(\log\log n)$ using $n/(\log\log n)$ processors of a CRCW PRAM. Vishkin [V91], presented an algorithm whose text analysis runs in time $O(\log^* n)$ using $n/\log^* n$ processors of a CRCW PRAM, and recently Galil [G92] presented an algorithm that runs in constant time and uses linear number of processors. This result has now been extended to the two-dimensional case [CCG+93] as well, using several new ideas.
2. Mathies [M88] had presented a parallel algorithm for solving the multi-dimensional pattern matching problem that runs in $O(d \log^2 n)$ time using n^d processors of a CRCW PRAM. Subsequently, Amir and Landau [AL88] presented an improved algorithm for the multi-dimensional pattern matching problem that runs in $O(d \log m)$ time using n^d processors of a CRCW PRAM. Previous work does not yield an optimal speedup parallel algorithm for this problem.

2. An optimal speedup algorithm for the s-p matching problem.

In this section, we start with a parallel algorithm for solving the s-p matching problem that embodies some of the main ideas, but does not have optimal speedup. Then, in the subsequent subsections, we progressively refine it by introducing additional techniques to eventually derive a parallel algorithm with optimal speedup, in Section 2.4.

2.1. Computing characteristics and thereby deriving δ. In the interests of completeness, we start with a formal definition of the *s-p matching problem*:

Input:. Strings $A = a_0 a_1 \ldots a_{m-1}$ and $B = b_0 b_1 \ldots b_{m-1}$ over an alphabet Σ.

Output:. A bit vector $\delta[0 \ldots m-1]$ where $\delta[i] = 1$ if and only if $a_{m-i-1} a_{m-i} \ldots a_{m-1} = b_0 b_1 \ldots b_i$.

The most straightforward computation of δ, which is used to solve the s-p matching problem, would involve explicit manipulation of the m suffixes of A and the m prefixes of B. However, this is obviously inefficient. This inefficiency can be overcome by observing that the set S of these $2m$ suffixes and prefixes splits into at most $2m$ (and at least m) equivalence classes under the relation of equality. Therefore, there exists a function that maps S into the set $[1 \ldots 2m]$ with the property that two string of same length are mapped onto the same value if and only if they are equal. It will be sufficient for us to compute *any* such function in order to determine δ.

Therefore, throughout the rest of this section, we will be concerned with computing such a characteristic function χ (or characteristic for short) for a given set of strings. In particular, we wish to compute a characteristic function that maps the elements of such a set to "small" integers, such that two elements of the set are mapped on the same integer if and only if they are equal. These χ values can then be used to quickly compute δ.

Without loss of generality, we will assume that m is a power two. To help in the explanation, we will start with the assumption that we are given $c \times m$ processors for an appropriately chosen constant c. (Essentially, we will assume that there is a processor for every character in the given input.) Subsequently, we will extend this algorithm to one that only uses $m/\log m$ processors by using Brent's lemma [Br74]; the optimal speedup result will follow from this extension.

2.2. A simple suboptimal algorithm. We first sketch a simple algorithm for computing δ in $O(m \log m)$ work and time $O(\log m)$. This algorithm computes the characteristics of the suffixes of A and the prefixes of B in S by appropriately combining the characteristics of their substrings.

Specifically, for each $i, i = 0, 1, \ldots, \log m$, we compute the characteristic of the set of all substrings of A and B of length 2^i, as was done in [AILSV88] and [KP92]. For purpose of illustration, we will describe this computation for a value i assuming that the characteristics were computed for $i-1$. We note that substrings of the same length are always handled concurrently. We first note that substrings of A and B are handled similarly here. Let $a_j a_{j+1} \ldots a_{j+2^i-1}$ be a substring of A. Dedicate a processor, say P_k, $(0 \leq k \leq 2m-1)$ to this substring. Given that the characteristics of strings of length 2^{i-1} were computed in the previous step, $\chi(a_j \ldots a_{j+2^{i-1}-1})$ and $\chi(a_{j+2^{i-1}} \ldots a_{j+2^i-1})$ are known and are in the range $[0 \ldots 2m-1]$. This information is now combined to compute the characteristics of substring $a_j a_{j+1} \ldots a_{j+2^i-1}$ as follows. Processor P_k writes k into location number $\chi(a_j \ldots a_{j+2^{i-1}-1}) + 2m\chi(a_{j+2^{i-1}} \ldots a_{j+2^i-1})$ of some vector indexed by $0, 1, \ldots, (2m)^2-1$. Then P_k reads the value in the above location and assigns this to $\chi(a_j, a_{j+1} \ldots a_{j+2^i-1})$. As all the processors write in parallel, only one of the processors writing into a location succeeds, and all processors writing into this location read that value. Moreover, the resulting χ value is in the range $[0 \ldots 2m-1]$. Note that in phase 0 the vector is of size m, the size of the alphabet; processor P_k writes k into location a_k in the vector.

These characteristics are now combined appropriately to derive the χ values of the suffixes and prefixes in S. Assume that the empty string is assigned the characteristic of $2m$ (to make it different from those computed above). Let $\bar{a}$ be a suffix of A of some length $\sum_{i=0}^{\log m} c_i 2^i$, $c_i \in \{0, 1\}$. Write $\bar{a}$ as the concatenation $\bar{a}_{\log m} \bar{a}_{\log(m-1)} \ldots \bar{a}_0$ where the length of $\bar{a}_i$ is $c_i 2^i$ (that is, the length is 0 or 2^i). Do the same for each prefix $\bar{b}$ of B. In $\log m$ steps it is possible to compute characteristics for the set S, by combining in step i, $i = 1, \ldots, \log m$, the values $\chi(\bar{a}_0 \ldots \bar{a}_{i-1})$ and $\chi(\bar{a}_i)$ to obtain $\chi(\bar{a}_0 \ldots \bar{a}_i)$. (To use the approach of the previous paragraph, replace $2m$ by $2m+1$.) All $\bar{a}$ and $\bar{b}$ are processed in parallel as above to assure consistent assignment of characteristics to strings in S.

Observation 1. *The function δ can be computed in $O(m \log m)$ work and time $O(\log m)$.*

2.3. Characterizing prefixes efficiently. In this section, we refine the above algorithm to where the total work done is only $O(m)$ on the string B for which the characteristics of the *prefixes* have to be computed. This computation will run in time $O(\log m)$. However, the corresponding characterization of the *suffixes* of string A will still need $O(m \log m)$ work. We will return to a discussion of this issue in Section 2.3.2 below. Subsequently, in Section 2.4, we will further refine these ideas to get an algorithm that solves the s-p matching problem in $O(\log m)$ time and $O(m)$ overall work.

In order to improve the work done in characterizing the prefixes of string B, we structure the computation to proceed in two stages, referred to respectively as the *winding* stage and the *unwinding* stage. The winding stage consists of phases $0, 1, \ldots, \log m$. In phase i, we compute the characteristic of the set of substrings $\{b_j \ldots b_{j+2^i-1} \mid 0 \leq j \leq m-2^i$ and j divisible by $2^i\}$. Basically, in phase i, only those positions whose distance from the head of B is a multiple of 2^i are "active". The computation corresponding to all the other positions will have "gone to sleep." For each active position j, we compute the characteristic of the substring of length 2^i starting at j. Therefore, on phase i, we compute the characteristics of only $m/2^i$ substrings of B [2].

In the (complementary) unwinding stage the information computed in the winding stage is combined. Its schedule is the "reverse" of that of the winding stage, and its phases are $\log m - 2, \log m - 3, \ldots, 0$. In phase i we compute the characteristics of the set $\{b_0 \ldots b_j \mid j > 2^i$, $j+1$ divisible by 2^i and not divisible by $2^{i+1}\}$. At this point, the characteristic of the prefix of B ending at position j is computed by combining previously computed characteristics of its substrings, as described in Section 2.2.

Remark: It is easily verified that the unwinding stage has two phases less than the winding stage. This is because the positions scheduled during phases $\log m$ and $\log m - 1$ in string B have their characteristics *completely* computed during the winding stage. Since only nodes whose characteristics are not completely computed during the winding stage need to be processed during the unwinding stage, we can drop the counterparts of these winding phases, during unwinding. Therefore, we start with an unwinding phase of $\log m - 2$.

We will also introduce an example below, to illustrate these issues.

2.3.1. An example. We will now present an example of the naming as it proceeds on the example strings $A=$ *cabacaba* and $B=$ *abacabab*. Essentially, in the algorithm described thus far, during the winding as well as the unwinding phases, the computation on A mimics the computation on B, so that the final χ values are correct. In other words, the algorithm must compute characteristics for suffixes of A that are *consistent* with that given to the prefixes of B. To do this, the intermediate steps in the computation of the characteristics of the suffix of A of any length j, must follow the the computation of the characteristic of the prefix of B of length j.

In Table 1 we show the schedule based on which positions are active during the winding and the unwinding stages of string B. At each phase, we also show the characteristic values computed for the active positions.

In Table 2, we illustrate the way in which positions in string A are active and "keep up" with the computation on string B. The final value of each position is shown in bold-face in Table 1 as

[2] The scheduling of computation on substrings of B is based on the algorithm from [KP92] used for computing characteristics of lineage functions of forests, and is similar to the well-known prefix sum computation [FL80]. Informally, we are given an input forest whose vertices and/or edges are labeled. A lineage function maps a set of labels of paths in this forest into some (range) set. However, since strings are a very special (degenerate) case of arbitrary forests, the techniques used here in the case of strings are significantly simpler than those used in the context of arbitrary forests for which the algorithm was originally designed.

			Winding	**Stage**		**Unwinding**	**Stage**
Position	*Processor Number*	*Phase 0*	*Phase 1*	*Phase 2*	*Phase 3*	*Phase 1*	*Phase 0*
		χ	χ	χ	χ	χ	χ
0	*8*	**12**					
1	*9*	*6*	**15**				
2	*10*	*12*					**10**
3	*11*	*4*	*11*	**11**			
4	*12*	*12*					**12**
5	*13*	*6*	*15*			**13**	
6	*14*	*12*					**14**
7	*15*	*6*	*15*	*15*	**15**		

Table 1
Execution trace on string $B = abacabab$.

			Winding	**Stage**		**Unwinding**	**Stage**
Position	*Processor Number*	*Phase 0*	*Phase 1*	*Phase 2*	*Phase 3*	*Phase 1*	*Phase 0*
		χ	χ	χ	χ	χ	χ
0	*0*	*4*	*0*	*4*	**0**		
1	*1*	*12*	*15*	*11*		*13*	**14**
2	*2*	*6*	*2*	*2*		**2**	
3	*3*	*12*	*11*	*3*			**3**
4	*4*	*4*	*0*	**4**			
5	*5*	*12*	*15*				**10**
6	*6*	*6*	**2**				
7	*7*	**12**					

Table 2
Execution trace on string $A = cabacaba$.

well as in Table 2. As noted in the remark above, we note that that the number of unwinding phases are two less than their winding counterparts. The results of the computations from Tables 1 and 2 are summarized below over all the phases in Table 3.

Let us consider a typical match of the prefix of B of size 7 with the suffix of A of the same length; the match is induced by the sequence $\sigma = abacaba$. Tracing through the tables, we see that these names are computed by decomposing σ into subcomputations on the three substrings as follows: $abac|ab|a$. Let us consider the unwinding stage in Tables 1 and 2 to understand this better. At the end of the winding stage, the characteristic of $abac$ is **11**, that of ab is **15** and that of the last symbol a in isolation is **12**. In Phase 1 of the unwinding stage, processors 13 and 1 characterized the string $abacab$ to be **13**. Finally, in Phase 0, processors 1 and 14 computed the final characteristic of **14** respectively in strings A and B, declaring the match.

It is easy to verify that the the resulting characteristics of two substrings are always the same, whenever they are identical. In the present implementation, we allow non-identical strings of different lengths to sometimes get the same characteristic value. However, this does not cause any problem in solving the s-p matching problem correctly.

2.3.2. The difficulty in characterizing suffixes efficiently. Recall that in the algorithm discussed above, during phase i of the winding stage, characteristics of substrings $\{a_j \ldots a_{j+2^i-1} \mid$

Phase 0:	a=12, b=6, c=4.
Phase 1:	ab=15, ac=11, ba=2, ca= 0.
Phase 2:	$abac$=11, $abab$=15, $acab$=3, $baca$= 2, $caba$= 4.
Phase 3:	$abacabab$=15, $cabacaba$=0.
Phase 1:	$abacab$=13, $bacaba$=2.
Phase 0:	aba=10, $abaca$=12, $acaba$=3, $abacaba$=14.

TABLE 3
Summary of characteristics of the substrings of A and B.

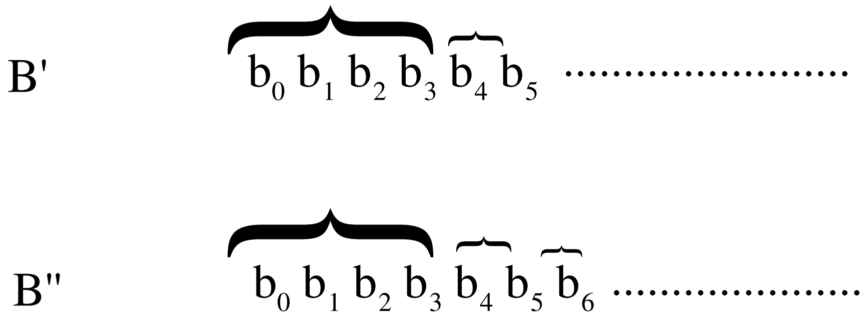
B'
$b_0\, b_1\, b_2\, b_3\, b_4\, b_5$ ....
B''
$b_0\, b_1\, b_2\, b_3\, b_4\, b_5\, b_6$ ....

FIG. 1. *Prefixes B' and B'' share intermediate substrings.*

$0 \leq j \leq m - 2^i\}$ of string A are computed. We observe that no more than $m - 2^i + 1$ substrings of A are active during this phase. Likewise, during phase i of the unwinding, for each k, and j such that $0 \leq k \leq m - 2^i + 1$, $j + 1$ is *maximal* and is divisible by 2^i and not divisible by 2^{i+1} the characteristics of the set $\{a_k \dots a_{k+j}\}$ of substrings of A are computed. For example consider *m=128, i=5.* Then k is in the range of $0 \dots 97$. Now, consider $k = 10$; we compute $j + 1$ to be 96, and therefore, the characteristics of $\{a_{10} \dots a_{105}\}$ will be computed. Once again, it is easy to verify that less than $m - 2^i + 1$ substrings of A are active during this phase.

Note that prefixes of B of lengths $\alpha 2^k + 1$ and $\alpha 2^k + 2$ for some α, share many overlapping substrings. Indeed, it was this fact that allowed us in Section 2.3 to structure the winding and unwinding stages such that the only $O(m)$ work was done on string B overall. However, as shown in the example below, the “simulation” of this computation on the suffixes of A does not have this nice structure. In particular, suffixes of increasing lengths of string A do not share overlapping substrings in such a simulation. As such, it is not hard to verify that by the end of the winding stage, we would have computed the characteristics of a set of $O(m \log m)$ substrings of A but of only $O(m)$ substrings of B. We will now sketch a brief example, to better illustrate this difficulty. In Figure 1, we have an example string B and we consider two prefixes of it, respectively of six characters and seven characters each and denoted by B' and B''. Similarly, we consider an example string A shown in Figure 2, and two of its suffixes in turn also respectively with six and seven characters are A' and A''.

Clearly, one possible case of s-p matching is where the suffix A' of A of length six is aligned with the prefix B' of B, of the same length. Similarly, the second case is where A'' is aligned with B''. As shown in the figures (and explained above), prefixes B' and B'' share intermediate substrings that are composed during the naming process. For example, the characteristic of B'' is derived simply by adding the single character b_6 to the characteristic of B'. However, as we can see from Figure 2, this is not true of the corresponding suffix A'', with respect to A'.

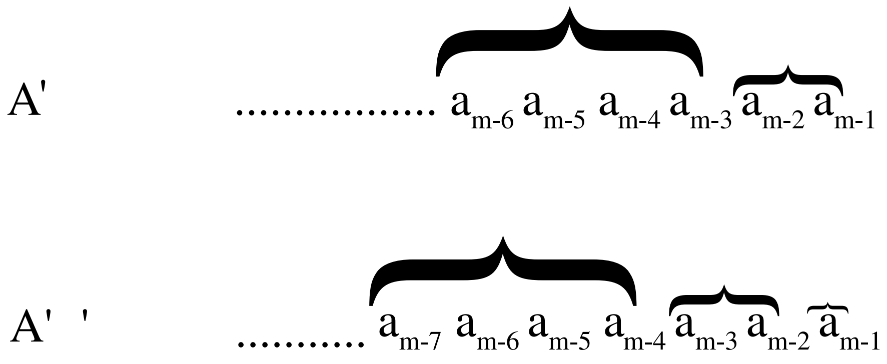
A'
.... a m-6 a m-5 a m-4 a m-3 a m-2 a m-1
A' '
.... a m-7 a m-6 a m-5 a m-4 a m-3 a m-2 a m-1

FIG. 2. *Suffixes A' and A'' do not share intermediate substrings.*

2.4. The optimal algorithm. Recall from the previous section that A was the "difficult" string as it required $O(m \log m)$ work. Intuitively, the way in which we get around this, is to shrink A to a string of size $m/\log m$. This is done by first partitioning it into $m/\log m$ non-overlapping substrings, each of length $\log m$. We then replace each substring by its characteristic value χ to get a new string $\bar{A}$. In conjunction with this, we decompose B into $\log m$ strings, $\bar{B}_1, \bar{B}_2, \ldots \bar{B}_{\log m}$, each of length $m/\log m$ as follows: character i in the jth such string characterizes the $\log m$ length substring of B starting at position $j + i \log m$. We operate on each of these $\log m$ cases generated by B independently, using the single copy of $\bar{A}$ as described in Section 2.3.

The algorithm is stated concisely in Section 2.4.1. This is followed by a detailed example in Section 2.4.2, that illustrates this concise description. In order to achieve parallel speedup, the algorithm for suffix-prefix matching being discussed here relies on a parallel construction of the well-known Aho-Corasick automation from [AC75]. We describe the details of this parallel construction in Section 2.4.3. This construction is used to implement Steps 1 and 2 from Section 2.4.1, as shown in Section 2.4.4. Subsequently, Step 3 of the algorithm is described in Sections 2.4.5 and 2.4.6. The details of coping with "boundary conditions" in Step 4 are discussed in Section 2.4.7, and solving the s-p matching problem in Step 5 is summarized in Section 2.4.8. Finally, the complexity of the overall algorithm is analyzed in Section 2.4.9.

2.4.1. Concise statement of the algorithm.

1. Compute in parallel the Aho-Corasick automaton M representing the set of $m/\log m$ non-overlapping substrings of A, of length $\log m$ each:

$$\begin{array}{l} a_0 a_1 \ldots a_{\log m - 1}, \\ a_{\log m} a_{\log m+1} \ldots a_{2\log m - 1} \\ \vdots \\ a_{m-\log m} a_{m - \log m + 1} \ldots a_{m-1}. \end{array}$$

   Let $\bar{a}_i$ for $i = 0, \log m, \ldots, m - \log m$ be the name (number) of the state *accepting* the substring $a_i a_{i+1} \ldots a_{i+\log m - 1}$. (We note that a state $\bar{a}_i$ for $i = 0, \log m, \ldots, m - \log m$ accepts the substring $a_i a_{i+1} \ldots a_{i+\log m-1}$ if and only if from the start state, the sequence of transitions induced by $a_i a_{i+1} \ldots a_{i+\log m - 1}$ lead to state $\bar{a}_i$.) Create the string $\bar{A} = \bar{a}_0 \bar{a}_{\log m} \bar{a}_{2\log m} \ldots \bar{a}_{m - \log m}$.

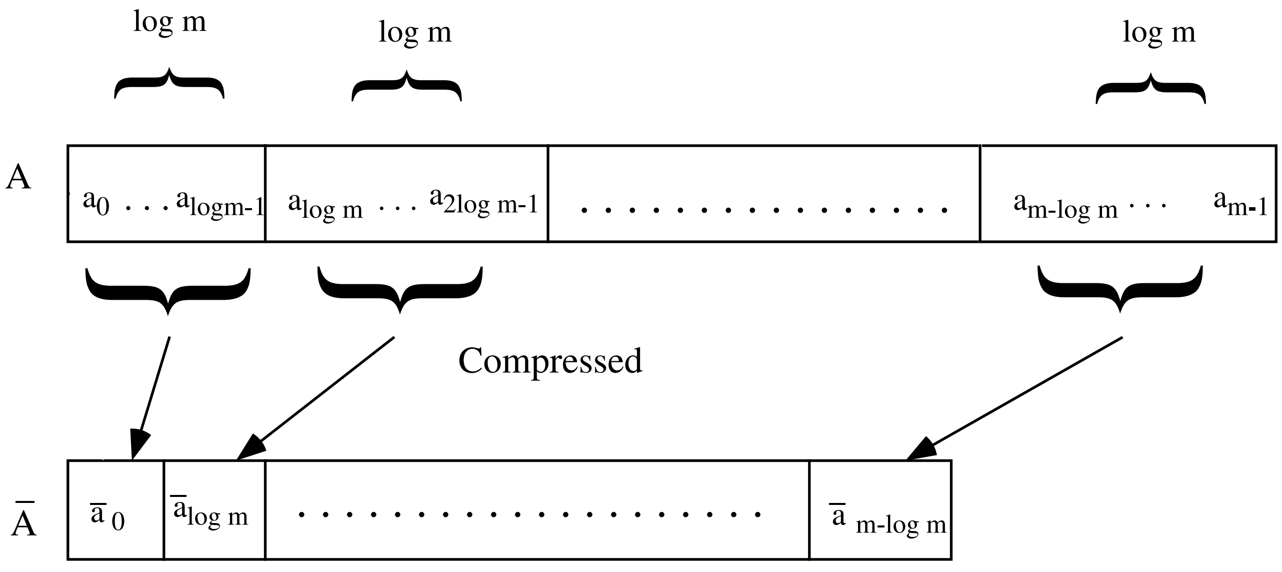

FIG. 3. *The transformation done to string A.*

2. Apply in parallel the automaton M to the string B considered as text. As the result, create the string $\bar{B} = \bar{b}_0\bar{b}_1 \ldots \bar{b}_{m-\log m}$ where $\bar{b}_i$ is the state accepting the string $b_i b_{i+1} \ldots b_{i+\log m-1}$. Create $\log m$ strings

$$\begin{aligned}
\bar{B}_1 &= \bar{b}_1\bar{b}_{\log m+1}\bar{b}_{2\log m+1} \cdots \bar{b}_{m-2\log m+1},\\
\bar{B}_2 &= \bar{b}_2\bar{b}_{\log m+2}\bar{b}_{2\log m+2} \cdots \bar{b}_{m-2\log m+2},\\
&\vdots\\
\bar{B}_{\log m} &= \bar{b}_{\log m}\bar{b}_{2\log m} \cdots \bar{b}_{m-\log m}.
\end{aligned}$$

   At this point, we have "compressed" string A by a factor of $\log m$, as shown in Figure 3.

   Furthermore, we have also "decomposed" string B into $\log m$ components, each of length $m/\log m$; one such decomposed component is illustrated in Figure 4.

3. Compute in parallel the characteristics of the set consisting of all the proper suffixes of the string $\bar{A}$ and all the proper prefixes of the strings $\bar{B}_1$, $\bar{B}_2$, ..., $\bar{B}_{\log m}$.
4. Compute in parallel the characteristics of the set consisting of all the suffixes of the set of strings

$$\begin{aligned}
&a_0a_1 \ldots a_{\log m-1},\\
&a_{\log m}a_{\log m+1} \cdots a_{2\log m-1}, \ldots,\\
&a_{m-\log m}a_{m-\log m+1} \cdots a_{m-1}
\end{aligned}$$

   and all the prefixes of the string $b_0b_1 \ldots b_{\log m-1}$.
   This part of the computation is used in processing the "remainder" pieces as shown in Figure 4.
5. Compute in parallel the vector δ. Let j, where $1 \le j \le m$, $j = c_1 \log m + c_2$, where $c_1 \ge 0$, and $1 \le c_2 \le \log m$. Also, let $i = j - 1$. Then $\delta[i] = 1$ if and only if:
   (a) $\chi(\bar{a}_{m-c_1\log m}\bar{a}_{m-(c_1-1)\log m} \cdots \bar{a}_{m-\log m}) = \chi(\bar{b}_{c_2}\bar{b}_{c_2+\log m} \cdots \bar{b}_{c_2+(c_1-1)\log m})$ and,
   (b) $\chi(a_{m-i-1}a_{m-i} \cdots a_{m-c_1\log m-1}) = \chi(b_0b_1 \ldots b_{c_2-1})$.

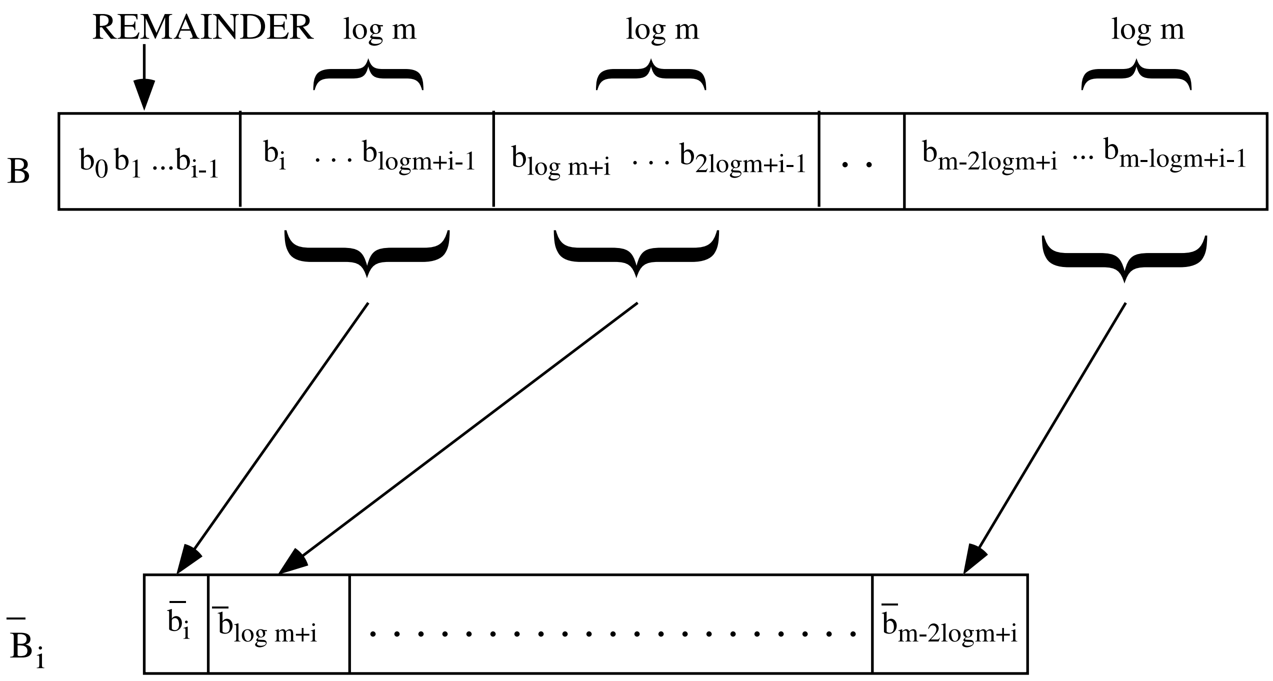

FIG. 4. *The manner in which string B decomposes into $\bar{B}_i$.*

2.4.2. Example. We now present an example. Let $\Sigma = \{a, b\}$, $m = 16$ and let

$$A = bbabbbaaabaabbab, \ B = aabaabbabababaaa$$

1. Decompose $A = bbab|bbaa|abaa|bbab$ (we broke A into strings of length $\log_2 16 = 4$ each). We now need to construct an automaton representing the 4 (actually 3 distinct) substrings. The automaton has 10 states. Its starting state is ϕ, and it is defined by the functions listed below, and represented in Figure 5. In this figure, the "goto" function is denoted by the solid lines, whereas the "failure" function is indicated by the dashed or broken lines; please refer to Section 2.4.3 for a detailed review of the structure of such an automaton.

$state$	$symbol$	$g(state, symbol)$
ϕ	a	α
ϕ	b	ϵ
α	a	
α	b	β
β	a	γ
β	b	
γ	a	δ
γ	b	
δ	a	
δ	b	
ϵ	a	
ϵ	b	ζ
ζ	a	η
ζ	b	
η	a	θ
η	b	ι
θ	a	
θ	b	
ι	a	
ι	b	

$state$	$f(state)$
ϕ	ϕ
α	ϕ
β	ϵ
γ	α
δ	α
ϵ	ϕ
ζ	ϵ
η	α
θ	α
ι	β

δ stands for *abaa*, θ for *bbaa*, ι for *bbab*. Thus, $\bar{A} = \iota\theta\delta\iota$

2.

$$\bar{B} = \gamma\delta\beta\zeta\eta\iota\gamma\beta\gamma\beta\gamma\delta\alpha$$

$$\bar{B}_1 = \delta\iota\beta,\ \bar{B}_2 = \beta\gamma\gamma,\ \bar{B}_3 = \zeta\beta\delta,\ \bar{B}_4 = \eta\gamma\alpha$$

3. The relevant (distinct) substrings consisting of certain suffixes of $\bar{A}$ and prefixes of $\bar{B}_1, \bar{B}_2, \bar{B}_3, \bar{B}_4$, are: β, δ, ζ, η, ι, $\beta\gamma$, $\delta\iota$, $\zeta\beta$, $\eta\gamma$, $\beta\gamma\gamma$, $\delta\iota\beta$, $\zeta\beta\delta$, $\eta\gamma\alpha$, $\theta\delta\iota$, and we may assume that characteristics have been computed for them appropriately.
4. The relevant (distinct) substrings are: *a*, *b*, *aa*, *ab*, *aab*, *baa*, *bab*, *aaba*, *abaa*, *bbaa*, *bbab*, and we may assume that characteristics have been computed for them appropriately.

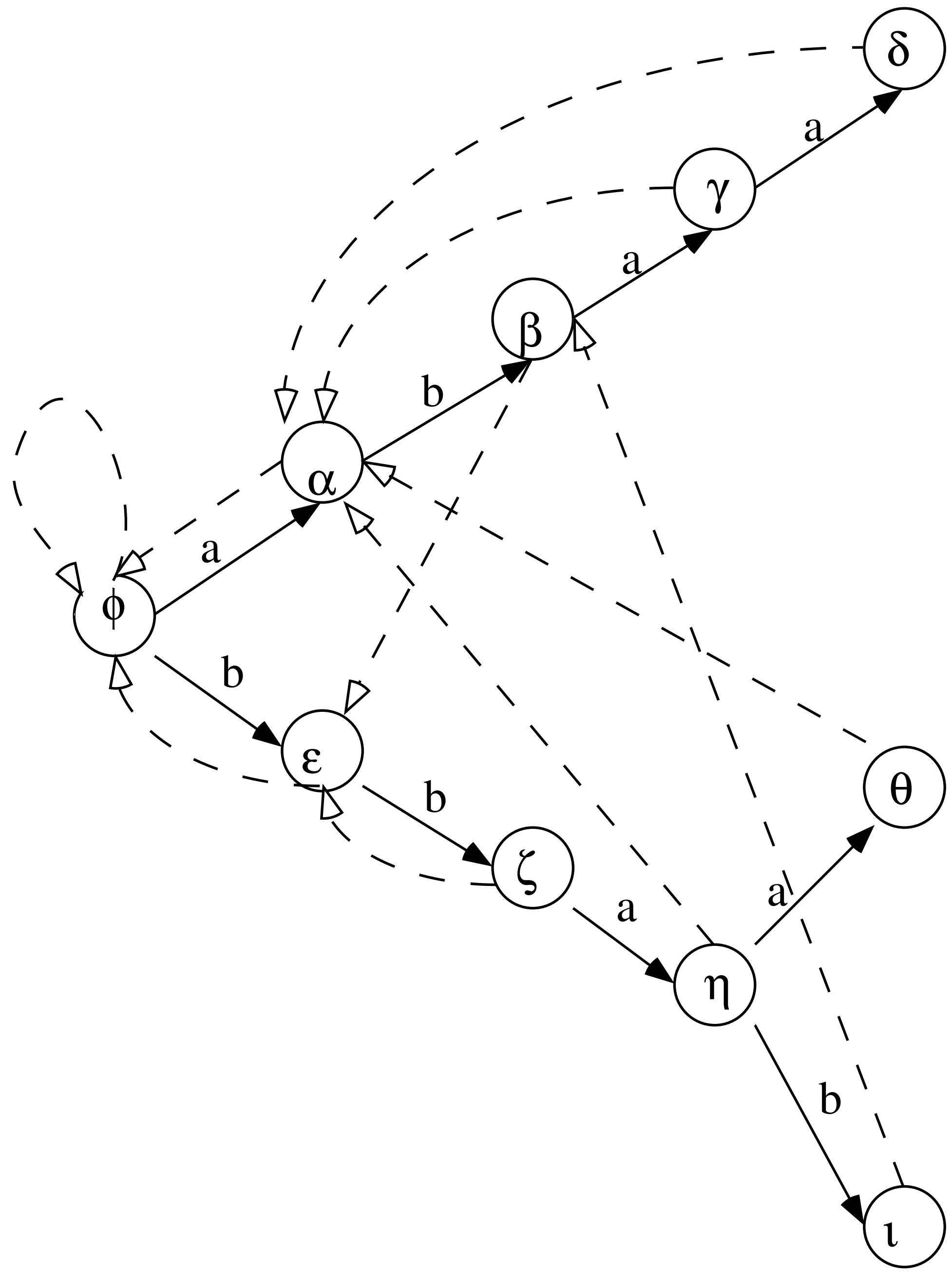

Fig. 5. *The Aho-Corasick automation for this example.*

5. Let us compute $\delta[8]$. $i = 8$, $j = i + 1 = 9$. Thus $j = 2 \cdot 4 + 1$, and therefore $c_1 = 2$ and $c_2 = 1$. Note that $\bar{a}_8\bar{a}_{12} = \bar{b}_1\bar{b}_5 = \delta\iota$; $\delta\iota$ stands for *abaabbab*. The equality of $\bar{a}_8\bar{a}_{12} = \bar{b}_1\bar{b}_5$ is determined by checking the characteristics of $\bar{a}_8\bar{a}_{12}$ and $\bar{b}_1\bar{b}_5$. Furthermore, $a_7 = b_0 = a$, also checked by characteristics. Thus $\delta[8] = 1$. Indeed the suffix of length 9 of A and the prefix of length 9 of B are both equal to *aabaabbab*.

Before proceeding any further, we will first describe the construction of the Aho-Corasick automaton in detail. It has to be recalled that this construction is done in Step 1 of the algorithm described above in Section 2.4.1.

2.4.3. The Aho-Corasick Automaton and its Parallel Construction. We assume that the reader is familiar with the work of Aho and Corasick from [AC75], to some extent. However, we will briefly review that work now, and fix the notation. Given several pattern strings, first a tree (trie) describing them is constructed. This tree describes the underlying automaton (see Figure 5), whose states correspond to the nodes. We have a "goto" function $g(state = r, symbol = a)$ represented by the solid lines in Figure 5 that indicates the next state to go-to, if we are in state r and the current symbol in the text string being read is a. In other words, $g(r, a) = s$ if the prefix can be extended by the symbol a. The new state will be s. Let $\eta(r)$ denote the *depth* of the state r, that is the distance between it and the root (ϕ in Figure 5) of the tree. The automaton is in state r if and only if:
(i) the suffix of the prefix of the text string examined so far is equal to the labels of the states from the root to r. Let α denote this sequence of labels to r.
(ii) Furthermore, α is equal to a prefix of some pattern string(s).
(iii) Also, $\eta(r)$ is the largest possible match found thus far, ending in the text-string position being currently matched.

We also have the "failure" function $f(r)$ represented by the broken lines in Figure 5. The failure function is used to continue matching the pattern on the string should $g(r, a) = fail$. This indicates that there is no pattern string whose prefix is $\alpha.a$ where . denotes concatenation as a prefix. Equivalently, any symbol of the text being read cannot be used to extend the prefix. That is if $g(r, a)$ for the symbol a that was read is undefined, the automaton goes to state $f(r)$ and processes a again. In other words, it checks whether $g(f(r), a)$ is defined. If it is, the transition takes place, otherwise, $f(f(r))$ is checked, and so on.

The parallel construction

We start by assigning a processor to each pattern string. Recall that, conceptually, each state of the automaton corresponds to a substring of one or more of the pattern strings. Informally, each state of the automaton is associated with an array of size $|\Sigma|$. [3] Let us suppose that the automaton has been constructed to include all states of depth d or less. Let r be a state at depth d. Then, if $g(r, a) = s$, the parallel algorithm first allocates a new array of size $|\Sigma|$ to represent s. This is done by the (one of) the processor(s) allocated to a pattern string whose prefix of length $d + 1$ defined state s. Following this step, this processor adds a pointer in the position corresponding to a (in the array representing r), to point to the array representing s. This pointer implements $g(r, a) = s$.

We will now address the question of computing the failure function f. However, before proceeding with this parallel construction, it is helpful to briefly review its classical sequential construction. The sequential algorithm computes f iteratively, based on the depth d of the state s. Let r be the

[3] Recall that in our paper, we are concerned with a $|\Sigma|$ that is polynomially bounded in the size of the input strings. Therefore, the automaton can be trivially implemented using a polynomial amount of space.

(unique) parent state of s in the corresponding tree representation. In particular, let $g(r, a) = s$. Informally, the iterative process involves following the failure functions towards the root of the trie, till a state s' is encountered such that:

1. the string of symbols representing the path from the root to $\sigma' = s'$ is a proper suffix of the sequence of symbols encoding the path from the root to state s and,
2. σ' is the longest sequence with the above property.

Equivalently, let $\eta(r) = d - 1$ and let $g(r, a) = s$. Let $r_1, r_2, \ldots, r_\nu$, $(\nu \geq 1)$, be the shortest sequence with the following properties: $r_1 = r$, $r_{i+1} = f(r_i)$ for $i \geq 1$ and $g(r_\nu, a)$ was defined. Then $f(s) = g(r_\nu, a)$.

Returning to the question of constructing the failure functions in parallel, we will now show that the parallel algorithm will proceed in at most $O(D)$ steps, where D denotes the length of the longest pattern string in the input [4] For any state x, we denote by $\tau(x)$ the step in which the computation of $f(x)$ is finished. We will show that:

THEOREM 2.1. *For any state s, $\tau(s) \leq 2\eta(s) - 1 - \eta(f(s))$.*

Proof Again, let r, s, a be such that $g(r, a) = s$. The computation of $f(s)$ will start, in the step following the step in which $f(r)$ was computed. By (informal) induction on the depth of a state, $f(r)$ is computed by step $2\eta(r) - 1 - \eta(f(r))$. We will show that it will end no later than in step $2\eta(s) - 1 - \eta(f(s))$.

During the computation the processor P allocated to s (technically this is one of the processors allocated to the string whose prefix of length $\eta(s)$ terminates at s) follows the sequence $f(r_1), f(r_2), \ldots, f(r_{\nu-1})$, $\nu \geq 1$, described above, with one step required for examining each f (and associated g). However, we need to ensure that processor P does not wait by more than a constant amount of time to determine each of the $f(r_i)$ for $1 \leq i \leq (\nu - 1)$, or else processor P might have to "wait" for the processor computing $f(r_i)$ to complete its computation.

We will therefore first prove the following crucial bound on the time, that relates $\tau(r_i)$ and $\tau(s)$. Specifically,

LEMMA 2.2. *For $1 \leq i \leq (\nu - 1)$, $\tau(r_i) \leq 2\eta(s) - \eta(f(s)) - \nu - i + 1$.*

Proof We first recall that $f(r_i) = r_{i+1}$ and that $\eta(r_i) > \eta(r_{i+1})$, by construction. Therefore, we note that $\eta(s), \eta(r_1), \eta(r_2), \ldots, \eta(r_{\nu-1})$ is a strictly decreasing sequence of integers. From the above, it immediately follows that

$$\eta(r_i) \quad \leq \quad \eta(s) - i \tag{1}$$

From the monotonicity of the depths of the sequence r_i, it also follows that

$$\eta(r_\nu) \quad \leq \quad \eta(r_i) - (\nu - i) \tag{2}$$

or, since $\eta(f(s)) = \eta(r_\nu) + 1$,

$$\eta(f(s)) - 1 \quad \leq \quad \eta(r_i) - (\nu - i) \tag{3}$$

Equivalently, replacing i with $i + 1$ we have

$$\eta(f(s)) - 1 \quad \leq \quad \eta(r_{i+1}) - (\nu - i - 1) \tag{4}$$

[4] In our actual application, all the pattern strings are of (equal) $D = \log m$ length.

From (4) above and the fact that $\eta(f(r_i)) = \eta(r_{i+1})$, it immediately follows that

(5) $$\eta(f(s)) + (\nu - i) - 2 \quad \leq \quad \eta(f(r_i))$$

By our induction on the depth of the state, please recall that

(6) $$\tau(r_i) \quad \leq \quad 2\eta(r_i) - 1 - \eta(f(r_i))$$

for $i < \nu$. Substituting in (6) for $\eta(r_i)$ and $\eta(f(r_i))$, respectively from (1) and (5) above simplifying we have,

(7) $$\tau(r_i) \quad \leq \quad 2\eta(s) - \eta(f(s)) - \nu - i + 1$$

From the above Lemma, we deduce that $f(r_1)$ is the last of the failure functions to be computed from the sequence $f(r_1), f(r_2) \ldots f(r_{\nu-1})$. Furthermore, it is computed by $T = 2\eta(s) - \eta(f(s)) - \nu$, derived by substituting $i = 1$. This implies that all the required $\nu - 1$ values of f are known by step T. Let us now recall that in order to compute $f(s)$, processor P follows the sequence $f(r_1), f(r_2), \ldots, f(r_{\nu-1})$, $\nu \geq 1$, described above, with one step required for examining each f (and associated g). Assuming that processor P starts the computation of $f(s)$ after time step T we conclude that this computation is completed by step $T + (\nu - 1) = 2\eta(s) - \eta(f(s)) - 1$, and the theorem is proved. □

2.4.4. Computing characteristics of substrings of length $\log m$**.** From the construction in Section 2.4.3, it is easy compute the characteristics of the set of $m/\log m$ substrings $\bar{a}_0, \bar{a}_{\log m}, \ldots, \bar{a}_{m-\log m}$ of A (used to derive $\bar{A}$) As these strings are all of equal (short) length of $\log m$ symbols, it follows from Theorem 2.1 that

Observation 2. *The Aho-Corasick automaton M, accepting strings $\bar{a}_0, \bar{a}_{\log m}, \ldots, \bar{a}_{m-\log m}$, can be constructed in work $O(m)$ and time $O(\log m)$.* The χ values for the substrings of A are simply the names of the states that accept them in the automaton.

We now proceed to assign characteristics to the $m - \log m$ substrings of B as follows. Dedicate a processor P_k (here $0 \leq k \leq m/\log m - 1$) to compute the characteristics $\bar{b}_{k\log m}, \bar{b}_{k\log m+1}, \ldots, \bar{b}_{(k+1)\log m-1}$ of the $\log m$ substrings

$$b_{k\log m} \ldots b_{(k+1)\log m-1}, b_{k\log m+1} \ldots b_{(k+1)\log m}, \ldots, b_{(k+1)\log m-1} \ldots b_{(k+2)\log m-2}.$$

This is easily done by running the automaton on the string $b_{k\log m} \ldots b_{(k+2)\log m-2}$ sequentially! If a substring of length $\log m$ is accepted, we characterize it by the name of the accepting state. If it is not accepted, we characterize with the name of the state the automaton reached after processing it. This sequential computation is done in $O(\log m)$ time per processor. (Note that we are not actually assigning distinct characteristics to the substrings of B that are not equal to any of the $m/\log m$ substrings of A. As it turns out, this does not affect the correctness of our algorithm.)

2.4.5. A sketch of one of the cases. Assume for now that we have computed the characteristics of the following set of $2m/\log m$ substrings of length $\log m$ of A and B: $\bar{a}_i = \chi(a_i \ldots a_{i+\log m-1})$ and $\bar{b}_i = \chi(b_i \ldots b_{i+\log m-1})$, where i is a positive integer multiple of $\log m$. $\bar{a}_i$ and $\bar{b}_i$ are symbols from a suitable alphabet, and can be encoded in $O(\log m)$ bits. Now, consider the two strings $\bar{A} = \bar{a}_0\bar{a}_{\log m}$ $\bar{a}_{2\log m} \ldots \bar{a}_{m-\log m}$ and $\bar{B} = \bar{b}_0\bar{b}_{\log m}$ $\bar{b}_{2\log m} \ldots \bar{b}_{m-\log m}$. These two strings are of length $m/\log m$ each. The characteristics of the suffixes and the prefixes defined by these two strings can therefore be computed in work $O((m/\log m)\log(m/\log m)) = O(m)$ using our algorithm from Section 2.3. This will give a *partial* solution to the problem for the original input strings A and B. Specifically, we solve the problem of computing the characteristics, but only for the suffixes of A and the prefixes of B whose lengths are divisible by $\log m$. Notice that in this computation $O(m)$ work was devoted to $\bar{A}$ derived from A, but only $O(m/\log m)$ work was devoted to $\bar{B}$ derived from B. The extension of the approach to handle the *complete* problem will balance the requirements so that $O(m)$ work is devoted both to strings derived by A and those derived from B as well.

2.4.6. Computing characteristics for the original inputs. To generalize from Section 2.4.5, assume that we have computed the characteristics of the following substrings of A and B, each of length $\log m$:

$b_i b_{i+1} \dots b_{i+logm-1}$, for $i = 0, 1, \dots, m - \log m$, and $a_i a_{i+1} \dots a_{i+logm-1}$, for i divisible by $\log m$. Denote:

$$\bar{b}_i = \chi(b_i b_{i+1} \dots b_{i+logm-1})$$

$$\bar{a}_i = \chi(a_i a_{i+1} \dots a_{i+logm-1})$$

We will now proceed to solve the original problem by considering suffixes of the single string $\bar{A}$ and prefixes of the $\log m$ strings derived from B of the form

$$\begin{aligned} \bar{B}_1 &= \bar{b}_1 \bar{b}_{\log m+1} \bar{b}_{2\log m+1} \dots \bar{b}_{m-2\log m+1}, \\ \bar{B}_2 &= \bar{b}_2 \bar{b}_{\log m+2} \bar{b}_{2\log m+2} \dots \bar{b}_{m-2\log m+2}, \\ &\vdots \\ \bar{B}_{\log m} &= \bar{b}_{\log m} \bar{b}_{2\log m} \dots \bar{b}_{m-\log m}. \end{aligned}$$

It can be verified that, disregarding "remainder" strings of length $\leq \log m$, (this is sketched in Section 2.4.7) every potential matching of a suffix in A with a prefix of B is covered by one of these $\log m$ strings. Therefore, we characterize the set of all suffixes of $\bar{A}$ and the prefixes of each of the $\log m$ strings $\bar{B}_1, \bar{B}_2, \dots, \bar{B}_{\log m}$. Furthermore, from the previous discussion, it follows immediately that

Observation 3. *The characteristics of $\bar{A}$ can be computed in $O(m)$, and those of each of the $\bar{B}_i$ in work that is linear in the size of $(O(m/\log m))$. Hence the total work done is $O(m)$.*

2.4.7. Computing characteristics of the remainders. In B there are only $\log m$ distinct "remainder" strings of the form $b_0 b_1 \dots b_x$ where $0 \leq x \leq \log m - 1$. In other words, these are all possible prefixes of the substring formed by the first $\log m$ positions of B. In A, there are a total of m possible remainders. To see this, let us consider A as being decomposed into $m/\log m$ disjoint substrings each of length $\log m$. Each of these substrings contributes exactly $\log m$ of its suffixes as possible remainders. For example, consider the substring $a_{c\log m} \dots a_{(c+1)\log m-1}$. In this case, the remainders are all of its suffixes including the substring itself. This computation can be viewed as solving $m/\log m$ s-p matching problems, one for each of the substrings of A and a unique pattern string from B. However, the pattern in this case is only $O(\log m)$ characters long. Therefore, to solve the problem in parallel, we need to only assign $\log^2 m$ processors; $\log m$ proccessors are assigned to each of the distinct prefixes of B. Consider one of the substrings of A in question, of length $\log m$. With a single processor, its suffixes are named using the techniques outlined in Section 2.3 in $O(\log m)$ time (and work). (Note that in Section 2.3 the computation of the prefixes is efficient while the computation of the suffixes is less efficient. One can easily reverse this, namely, design an algorithm that has linear work on the suffix computation and $O(m \log m)$ work on the prefix computation.) Therefore, OBSERVATION 4. *the characteristics of the set of the remainder strings can be computed in $O(m)$ work and $O(\log m)$ time using up to $m/\log m$ processors.*

2.4.8. s-p matching from characteristics. We use characteristics to compute the vector δ.

Let $j \in \{1, 2, \ldots, m\}$ and we write $j = c_1 \log m + c_2$ where $c_1 \geq 0$, and $1 \leq c_2 \leq \log m$. Also, let $i = j - 1$. Then, $\delta[i] = 1$ if and only if

1. $a_{m-c_1 \log m} a_{m-c_1 \log m+1} \ldots a_{m-1} = b_{c_2} b_{c_2+1} \ldots b_{i=c_2-1+c_1 \log m}$ and,
2. $a_{m-i-1} a_{m-i} \ldots a_{m-c_1 \log m-1} = b_0 b_1 \ldots b_{c_2-1}$.

It is easy to see that the first condition is equivalent to verifying the following relationship between characteristics:

$$\chi(\bar{a}_{m-c_1 \log m} \bar{a}_{m-(c_1-1) \log m} \cdots \bar{a}_{m-\log m}) = \chi(\bar{b}_{c_2} \bar{b}_{c_2+\log m} \cdots \bar{b}_{c_2+(c_1-1) \log m}).$$

The condition checks whether the suffix of length c_1 of $\bar{A}$ (of length $c_1 \log m$ in A) is equal to the prefix of length c_1 of $\bar{B}_{c_2}$ (substring of length $c_1 \log m$ of B that starts with b_{c_2}). The second condition checks whether the remainders of length c_2 in both A and B are equal.

2.4.9. Complexity. We first state the time/processor complexity of the algorithm. The proof of the theorem is not given here as it follows in a straight-forward manner from the statement of the algorithm and the discussion and observations in the previous sections.

THEOREM 2.3.

The above algorithm solves the s-p matching problem in $O(\log m)$ time using work of $O(m)$ on a CRCW PRAM, given input strings A and B of size m each.

Given this theorem and using Brent's lemma, we immediately obtain an algorithm that solves the s-p matching problem in $O(\log m)$ time using $m/\log m$ processors (of a CRCW PRAM as well), given input strings A and B of size m each. Starting with Section 3, we will use this modified algorithm

To reiterate, our algorithm relies on two basic ideas. These ideas are:

1. computation of characteristics and
2. the usage of failure functions for recognizing very short strings of equal length by means of the Aho-Corasick automaton.

Using the first idea alone will give a sub-optimal speedup algorithm of work $O(m \log \log m)$. This is because we can replace the Aho-Corasick automaton and its application in naming the strings in Steps 1 and 2 of our algorithm (Section 2.4.1) with a procedure for computing characteristics. It is a simple exercise to verify that this replacement will require an additional $O(\log \log m)$ multiplicative work overhead, due to the computation on B.

3. Applying the s-p matching algorithm.

We will now describe various applications of our algorithm for s-p matching.

3.1. Multiple s-p matching problems. Most of the subsequent algorithms can be put in a setting of simultaneously solving the matching of several suffixes against several prefixes. Say we are given u text strings $T_1, T_2, \ldots, T_u$, respectively of lengths $n_1, n_2, \ldots, n_u$, and v patterns $P_1, P_2, \ldots, P_v$, each of length $m \leq n_i$ for $1 \leq i \leq u$. We wish to determine which suffixes of the T_i's match which prefixes of the P_j's. It is possible to formulate an algorithm for this general problem at this point, but instead we will develop various special cases of it as needed.

3.2. String matching. The classical string matching problem is defined by:

Input:. A *pattern* string of length m and a *text* string of length $n \geq m$.

Output:. All the positions in the text in which the pattern matches.

We will show how to reduce the solution of the problem to the solution of a version of multiple s-p matching problem.

Let the pattern string be $P = p_1p_2\dots p_m$ and let the text string be $T = t_1t_2\dots t_n$. "Cut" T into $\lceil n/m \rceil$ non-overlapping substrings $T_1, T_2, \dots, T_{\lceil n/m \rceil}$. $T_j = t_{(j-1)m+1}t_{(j-1)m+2}\dots t_{jm}$ for $j < \lceil n/m \rceil$ and $T_{\lceil n/m \rceil} = t_{\lceil n/m \rceil}t_{\lceil n/m \rceil+1}\dots t_n$. Thus, $T = T_1T_2\dots T_{\lceil n/m \rceil}$.

It is easy to see that the pattern matches some position in the text if and only if for some j, a suffix of length $k > 0$ of T_j matches a prefix of P and a prefix of length $m - k$ of T_{j+1} matches a suffix of P. If $k = m$ or $j = \lceil n/m \rceil$ then $j + 1$ is undefined. This observation can be used to immediately produce an algorithm for string matching. For a simple example consider: $P = abaa$, $T = babaaaaabaa$. Here, $T_1 = baba$, $T_2 = aaaa$, $T_3 = baa$.

P matches T in position 2 because:

1. aba is both a suffix of T_1 and a prefix of P.
2. a is both a prefix of T_2 and a suffix of P.

Thus, in general, we need to solve two subproblems:

1. The s-p matching problems for finding the matches between all the suffixes of the strings $T_1, T_2, \dots, T_{\lceil n/m \rceil}$ and the prefixes of the string P.
2. The s-p matching problems for finding the matches between all the suffixes of the string P and the prefixes of the strings $T_1, T_2, \dots, T_{\lceil n/m \rceil}$.

These two subproblems can be solved in time $O(\log m)$ and work $O(n)$ by a straightforward application of the standard s-p algorithm. For instance, the first subproblem, can be solved by *parallel* solution of the $\lceil n/m \rceil$ s-p problems, each defined by a pair of strings (T_j, P). This will characterize the set consisting of all the suffixes of $T_1, T_2, \dots, T_{\lceil n/m \rceil}$ and the prefixes of P. ($T_{\lceil n/m \rceil}$ could in general be shorter than m, but this is not significant.) Again, following Brent's lemma:

THEOREM 3.1.

Given a text string of length n and a pattern of length m, the above algorithm solves the string matching problem using $n/\log m$ processors in $O(\log m)$ time of a CRCW PRAM.

3.3. Multi-pattern string matching. We will now define the multi-pattern string matching problem.

Input:. A *text* string of length n and v *patterns* each of length m. (The patterns are not necessarily distinct.)

Output:. For each position in the text indicate if a pattern matches there. If a match exists, report one of the matching patterns.

Let T be the text string, and let $P_1, P_2, \ldots P_v$ be the patterns. Again consider an example first. Let

$$T = abbbab$$
$$P_1 = P_2 = ab,\ P_3 = ba.$$

Then ab matches T at positions 1 and 5, and ba matches T at position 4. Thus there are 4 "acceptable" representations of the output depending on whether we state that P_1 or P_2 match at positions 1 or at position 5. So if 0 indicates no match, the answer could be given by any of the four strings: $10031, 10032, 20031, 20032$. To help in making the presentation easier, in the rest of the paper, we will represent outputs in which all the occurrences of a substring corresponding to more than one pattern be denoted by a single symbol. Thus we would allow 10031 or 20032; however, 10032 and 20031 are disallowed as valid representations of the output. We will therefore require that the output is in a *canonical* form:

Output:. A string $Q = q_1 q_2 \ldots q_{n-m+1}$ over the alphabet $\{0, 1, \ldots, v\}$, satisfying the conditions:

1. $q_i = j > 0$ if P_j matches T at position i. $q_i = 0$ if there is no match.
2. $q_{i_1} = q_{i_2}$ if and only if P_{j_1} matches T at position i_1, P_{j_2} matches T at position i_2 and $P_{j_1} = P_{j_2}$.

For ease of exposition, assume, without loss of generality, that m divides n, and let $q = n/m$. Furthermore, we assume that all the patterns are distinct. We do this to obtain the output in normal form. There is no loss of generality in making this claim, since given a set of patterns $P_1, P_2, \ldots P_v$, we can eliminate duplicates and compute the reduced set consisting of only distinct patterns easily using naming (as in the case of string A before), in linear work and time $O(\log m)$.

Again cut the text T into q non-overlapping pieces of length m each: $T_1, T_2, \ldots, T_q$. Compute the characteristics of the set consisting of all the prefixes and all the suffixes of the set of strings $\Delta = \{P_1, P_2, \ldots, P_v, T_1, T_2, \ldots, T_q\}$. To derive linear work, the following algorithm is used:

1. Each string in Δ is cut into $m/\log m$ non-overlapping substrings of length $\log m$. Construct the automaton accepting all these substrings. Each substring is characterized by the name of the state accepting it. Compress the strings, obtaining $(v+q)$ strings of length $m/\log m$ each. Here each string in Δ plays the role of A in the s-p algorithm; this step is analogous to step 1 of the s-p algorithm.
2. For each string in Δ create $\log m$ strings of length $m/\log m - 1$ each. A symbol in a new string, stands for a substring of length $\log m$ in the original string. We obtain $(v+q)\log m$ strings of length $m/\log m - 1$ each. Here each string in Δ plays the role of B in the s-p algorithm; this step is analogous to step 2 of the s-p algorithm.
3. Compute the characteristics of the set consisting of all the proper suffixes of the strings computed in step 1 and all the proper prefixes of the strings computed in step 2. This step is analogous to step 3 of the s-p algorithm.
4. Compute the characteristics of the "remainder" strings. This step is analogous to step 4 of the s-p algorithm.
5. Generalizing from our pattern matching algorithm we note that P_i matches T at position $j = \alpha m - \beta$, $1 \le \alpha \le q$, $0 \le \beta \le m-1$, $1 \le j \le n-m+1$ if and only if the suffix of length $(\beta+1)$ of T_α matches the prefix of length $(\beta+1)$ of P_i and the prefix of length $(m-\beta-1)$ of $T_{\alpha+1}$ matches the suffix of length $(m-\beta-1)$ of P_i. This can be done, as sketched below. Assume the existence of some vector V of length $(2(n+vm)+1)^2$ initialized to 0. For a string S, let $pref(S,i)$ and $suf(S,i)$ denote respectively the prefix and the suffix of length

i of S.
From the previous steps, we can assume that we have computed the characteristics of all the prefixes and all the suffixes of the patterns and the text pieces. We proceed now in two steps:

(a) We assign a processor to each position of each pattern. In parallel, for each position $\gamma \leq m$ of each pattern P_i, the processor assigned to it writes i in the location $V[\chi(pref(P_i, \gamma)) + (2(n + vm) + 1)\chi(suf(P_i, m - \gamma))]$. As the result of this step, we have "coded" all existing pairs of prefix-suffix for each pattern.

(b) We assign a processor to each position of each T_j. The processor at position $m-\gamma+1$ of T_j reads the value of $V[\chi(suf(T_j, \gamma)) + (2(n + vm) + 1)\chi(pref(T_{j+1}, m - \gamma))]$. Pattern P_i matches T at position $jm - \gamma + 1$ if and only if the value read was i.

Step 5.a was needed to make sure that a spurious match is not obtained by combining a prefix of one pattern with the suffix of another pattern.

THEOREM 3.2.

Given a text string of length n and v patterns each of length m, the above algorithm solves the multi-pattern string matching problem using $(n + vm)/\log m$ processors in $O(\log m)$ time of a CRCW PRAM.

3.4. Multi-text/multi-pattern problem.

Input:. u *text* strings and v *patterns.* The length of the ith text string is n_i, and the length of all patterns m, is the same.

Output:.u strings $Q_1, Q_2, \ldots, Q_u$ over the alphabet $\{0, 1, \ldots, v\}$. Each $Q_j = q_{j,1}q_{j,2}\cdots q_{j,n_j-m+1}$ is of length $n_j - m + 1$ and satisfies the condition

1. $q_{k,i} = j > 0$ if and only if P_j matches T_k at position i. (0 indicates no match.)
2. if P_{j_1} matches T_{k_1} at position i_1 and P_{j_2} matches T_{k_2} at position i_2 and $P_{j_1} = P_{j_2}$, then $q_{k_1,i_1} = q_{k_2,i_2}$.

THEOREM 3.3.

The obvious modification of the algorithm in Section 3.3 solves the multi-text/multi-pattern matching problem in time $O(\log m)$ using $(vm + \sum_{j=1}^{u} n_j)/\log m$ processors.

3.5. Multi-dimensional pattern matching. We now describe our parallel algorithm for the multi-dimensional pattern matching problem.

Input:. A d-dimensional pattern array $P[1..m, 1..m, \ldots, 1..m]$ of size m^d and a d-dimensional text array $T[1..n, 1..n, \ldots, 1..n]$ of size n^d. ($P[1..m, 1..m, \ldots, 1..m]$ stands for $\{P[i_1, i_2, \ldots, i_d] \mid 1 \leq i_1, i_2, \ldots, i_d \leq m\}$, etc.)

Output:. A d-dimensional array $Q[1..n - m + 1, 1..n - m + 1, \ldots, 1..n - m + 1]$ over $\{0, 1\}$ such that $Q[i_1, i_2, \ldots, i_d] = 1$ if and only if the pattern matches the text in position $(i_1, i_2, \ldots, i_d)$, that is $T[i_1..i_1 + m - 1, i_2..i_2 + m - 1, \ldots, i_d..i_d + m - 1] = P[1..m, 1..m, \ldots, 1..m]$.

Essentially, we use the same framework as given in [Ba78, Bi77, KR87] in the sequential case. The sequential algorithm requires time of $O(dn^d)$. In [AL88] a parallel algorithm for this problem was given requiring work of $dn^d \log m$ and time of $O(d \log m)$. The improvement in complexity in the algorithm implementation we will present, comes from the fact that now we overcome the bottleneck in earlier algorithms by using our optimal algorithm for multi-text/multi-pattern string matching from Section 3.4

We describe the algorithm recursively:

1. Match the set of strings of the form $P[i_1, i_2, \ldots, i_{d-1}, 1..m]$ in the set of strings of the form $T[k_1, k_2, \ldots, k_{d-1}, j..j+m-1]$. Present the canonical output as several $(d-1)$-dimensional arrays. Specifically, we get the following arrays:
   (a) $R[1..m, 1..m, \ldots, 1..m]$. $R[i_1, i_2, \ldots, i_{d-1}]$ is $\chi(P[i_1, i_2, \ldots i_, i_{d-1}, 1..m])$
   (b) $S_j[1..n, 1..n, \ldots, 1..n]$, $j = 1, 2, \ldots, n-m+1$. $S_j[k_1, k_2, \ldots, k_{d-1}]$ is the characteristic of the string from P matching T at position $(k_1, k_2, \ldots, k_{d-1}, j)$, if such pattern exists; it is 0 if no such pattern exists.
2. Recursively solve several $(d-1)$-dimensional problems. Specifically, match R in $S_1, S_2, \ldots, S_{n-m+1}$.
   P matches T at position $T[k_1, k_2, \ldots, k_{d-1}, j]$ if and only if R matches S_j at position $k_1, k_2, \ldots, k_{d-1}$.

We now proceed to analyze the complexity of the algorithm. From section 3.4 we know that

FACT 1. *v pattern strings of length m each can be matched in $u \geq v$ text strings of length $n \geq m$ each in work $c \times u \times n$ for an appropriate constant c.* Using this fact, we need to show that

LEMMA 3.4. *For the above algorithm, $W_d(n^d)$, the work required by the text of size n^d, satisfies $W_d(n^d) \leq cdn^d$ and the time required is $O(d \log n)$.*

Proof Let us review the two steps above. In the first step we match m^{d-1} pattern strings of length m each in n^{d-1} text strings of length n each. This can be done in work of $cn^{d-1}n = cn^d$ and time of $O(\log n)$.

In the second step we solve in parallel $n-m+1$, $(d-1)$-dimensional problems each consisting of matching a pattern of size m^{d-1} in text of size n^{d-1}. By induction and the previous fact, this can be done in work $(n-m+1)W_{d-1}(n^{d-1}) \leq nc(d-1)n^{d-1} = c(d-1)n^d$. The time required is $O((d-1)\log n)$. Combining the complexities of the two steps we obtain that the work is $W_d(n^d) \leq cdn^d$ and the time is $O(d \log n)$. □

By interpreting the above lemma appropriately, we have

THEOREM 3.5. *The above algorithm solves the multi-dimensional pattern matching problem in time $O(d \log m)$ using $n^d / \log m$ processors of a CRCW PRAM.*

We now present an example. In this example, $d = 3$, $m = 2$, and $n = 3$. The problem instance is defined by:

$$P[1..2, 1..2, 1] = \begin{matrix} a & b \\ b & b \end{matrix}$$

$$P[1..2, 1..2, 2] = \begin{matrix} b & a \\ b & b \end{matrix}$$

$$T[1..3,1..3,1] = \begin{array}{ccc} a & b & a \\ b & a & a \\ b & b & b \end{array}$$

$$T[1..3,1..3,2] = \begin{array}{ccc} b & a & b \\ a & b & b \\ a & b & b \end{array}$$

$$T[1..3,1..3,3] = \begin{array}{ccc} a & b & a \\ b & b & b \\ a & a & a \end{array}$$

(Note that P matches T at position $(1,2,2)$).

The recursion has 3 stages:

1. We compute the characteristics of the appropriate substrings of P. Without loss of generality, they are:

$$\begin{array}{ll} P[1,1,1..2] = ab, & \chi(P[1,1,1..2]) = 1 \\ P[1,2,1..2] = ba, & \chi(P[1,2,1..2]) = 2 \\ P[2,1,1..2] = bb, & \chi(P[2,1,1..2]) = 3 \\ P[2,2,1..2] = bb, & \chi(P[2,2,1..2]) = 3 \end{array}$$

   We obtain the following characteristics for the strings of T:

$$\begin{array}{lll} T[1,1,1..3] = aba, & \chi(T[1,1,1..2]) = 1, & \chi(T[1,1,2..3]) = 2 \\ T[1,2,1..3] = bab, & \chi(T[1,2,1..2]) = 2, & \chi(T[1,2,2..3]) = 1 \\ T[1,3,1..3] = aba, & \chi(T[1,3,1..2]) = 1, & \chi(T[1,3,2..3]) = 2 \\ T[2,1,1..3] = bab, & \chi(T[2,1,1..2]) = 2, & \chi(T[2,1,2..3]) = 1 \\ T[2,2,1..3] = abb, & \chi(T[2,2,1..2]) = 1, & \chi(T[2,2,2..3]) = 3 \\ T[2,3,1..3] = abb, & \chi(T[2,3,1..2]) = 1, & \chi(T[2,3,2..3]) = 3 \\ T[3,1,1..3] = baa, & \chi(T[3,1,1..2]) = 2, & \chi(T[3,1,2..3]) = 0 \\ T[3,2,1..3] = bba, & \chi(T[3,2,1..2]) = 3, & \chi(T[3,2,2..3]) = 2 \\ T[3,3,1..3] = bba, & \chi(T[3,3,1..2]) = 3, & \chi(T[3,3,2..3]) = 2 \end{array}$$

   The pattern is now coded as a two-dimensional object $R[1..2,1..2]$; $R[i,j]$ stands for the characteristic of $P[i,j,1..2]$. The text is coded as two two-dimensional objects $S_1[1..3,1..3]$ and $S_2[1..3,1..3]$; $S_k[i,j]$ stands for the characteristic of $T[i,j,k..k+m-1]$.

$$R[1..2,1..2] = \begin{array}{cc} 1 & 2 \\ 3 & 3 \end{array}$$

$$S_1[1..3,1..3] = \begin{array}{ccc} 1 & 2 & 1 \\ 2 & 1 & 1 \\ 2 & 3 & 3 \end{array}$$

$$S_2[1..3, 1..3] = \begin{array}{ccc} 2 & 1 & 2 \\ 1 & 3 & 3 \\ 0 & 2 & 2 \end{array}$$

(Note that R matches S_2 at position $(1, 2)$.)

2. As stated in the description of the algorithm we should now solve two matching problems: R in S_1 and R in S_2. It is simpler to combine these two problems into the single problem of matching R in S_1 and S_2.
We compute the characteristics of the appropriate substrings of R. Without loss of generality they are:

$$\begin{array}{ll} R[1, 1..2] = 12, & \chi(R[1, 1..2]) = 1 \\ R[2, 1..2] = 33, & \chi(R[2, 1..2]) = 2 \end{array}$$

We obtain the following characteristics for the strings of S_1 and S_2:

$$\begin{array}{lll} S_1[1, 1..3] = 121, & \chi(S_1[1, 1..2]) = 1, & \chi(S_1[1, 2..3]) = 0 \\ S_1[2, 1..3] = 211, & \chi(S_1[2, 1..2]) = 0, & \chi(S_1[2, 2..3]) = 0 \\ S_1[3, 1..3] = 233, & \chi(S_1[3, 1..2]) = 0, & \chi(S_1[3, 2..3]) = 2 \\ S_2[1, 1..3] = 212, & \chi(S_2[1, 1..2]) = 0, & \chi(S_2[1, 2..3]) = 1 \\ S_2[2, 1..3] = 133, & \chi(S_2[2, 1..2]) = 0, & \chi(S_2[2, 2..3]) = 2 \\ S_2[3, 1..3] = 022, & \chi(S_2[3, 1..2]) = 0, & \chi(S_2[3, 2..3]) = 0 \end{array}$$

The pattern is now coded as a one-dimensional object $U[1..2]$; $U[i]$ stands for the characteristic of $R[i, 1..2]$. The text is coded as four one-dimensional objects $V_{1,1}[1..3], V_{1,2}[1..3], V_{2,1}[1..3], V_{2,2}[1..3]$; $V_{j,k}[i]$ stands for the characteristic of $S_j[i, k..k+1]$. (U and V play the same role as R and S in the previous stage.)

$$U[1..2] = 12$$

$$V_{1,1}[1..3] = 100$$

$$V_{1,2}[1..3] = 002$$

$$V_{2,1}[1..3] = 000$$

$$V_{2,2}[1..3] = 120$$

(Note that U matches $V_{2,2}$ at position 1.)

3. We should now solve four matching problems. Again we combine them into a single problem. We compute the characteristics of the appropriate substrings of U. Without loss of generality they are:

$$U[1..2] = 12, \quad \chi(U[1..2]) = 1$$

We obtain the following characteristics for the strings of $V_{1,1}, V_{1,2}, V_{2,1}, V_{2,2}$:

$$\begin{array}{lll} V_{1,1}[1..3] = 100, & \chi(V_{1,1}[1..2]) = 0, & \chi(V_{1,1}[2..3]) = 0 \\ V_{1,2}[1..3] = 002, & \chi(V_{1,2}[1..2]) = 0, & \chi(V_{1,2}[2..3]) = 0 \\ V_{2,1}[1..3] = 000, & \chi(V_{2,1}[1..2]) = 0, & \chi(V_{2,1}[2..3]) = 0 \\ V_{2,2}[1..3] = 120, & \chi(V_{2,2}[1..2]) = 1, & \chi(V_{2,2}[2..3]) = 0 \end{array}$$

In stage 3 we found that U matches $V_{2,2}$ at position 1. Therefore, the output of stage 2 is that R matches S_2 at position (1,2). Hence, in stage 1, we indeed deduce that P matches T at position $(1, 2, 2)$. Of course, the answer could be written in the form of Q, as required by the formal specifications of the output.

3.6. The pattern occurrence detection. In this section we consider the problem of string matching when the text has been cut into a number of pieces. Formally, we have:

Input:. A *pattern* string $P = p_0p_1 \dots p_{m-1}$ of length m and l *distinct text substrings* $T_1, T_2, \dots, T_l$ of length k each.

Output:. Decision whether there exists a permutation of the text substrings for which the pattern is a match, and if yes, produce one such permutation and the match position for it.

Before we proceed with the description of our algorithm, several remarks are in order. Generally, one might consider that the strings are of arbitrary lengths and possibly replicated. Then the "assembling" of strings into a single text string is difficult. The deterministic problem of pattern matching in this case is NP-hard [TU88]. Approximation algorithms for this more difficult problem, which appeared in molecular biology [TU88], were given in [CD88, TU88, KM95, Tur89, Ukk90, BJLTY91].

We now proceed with the description of our algorithm solving the simpler problem we have formally defined above. To simplify the exposition, consider the case when $m \geq 3k$ and m is a multiplication of k. Observe that a matching permutation exists if and only if there exist *distinct* $j_1, j_2, \dots, j_s$, where $j_s \geq 3$ such that the pattern is equal to a suffix of T_{j_1} followed by $T_{j_2}, \dots, T_{j_{s-1}}$, followed by a prefix of T_{j_s}.

The algorithm proceeds in two steps.

1. We compute an integer vector $CENTER[0, .., m-1]$.
   $CENTER[i] = j$ if and only if $p_ip_{i+1} \dots p_{i+k-1} = t_{j,0}t_{j,1} \dots t_{j,k-1}$; if no such j exists set $CENTER[i] = 0$.
2. We compute a bit vector $ENDS[0..k-1]$. $ENDS[0] = 1$. For $i \geq 1$, $ENDS[i] = 1$ if and only if there exist distinct j_a and j_b such that:
   (a) $suf(T_{j_a}, i) = pref(P, i)$ ($p_0 \dots p_{i-1}$ is equal to the suffix of length i of T_{j_a}.)
   (b) $pref(T_{j_b}, k-i) = suf(P, k-i)$
   (c) For each $r = 0, 1, \dots, m/k-2$, $p_{i+rk} \dots p_{i+(r+1)k-1}$ is different from both T_{j_a} and T_{j_b}.

A match exists if and only if for some $i = 0, 1, \dots, k-1$, (i) $ENDS[i] = 1$, (ii) all $CENTER[i+rk]$, for $r = 0, 1, \dots, m/k-2$, are $\neq 0$ and distinct.

All these vectors can be computed in time $O(\log m)$ using $O(kl)$ work. It is easy to see how to complete the algorithm to produce the output in those complexity bounds too.

If $m < k$ we also have to test whether the pattern matches one of the text pieces, which too can be done in optimal speedup. Therefore,

THEOREM 3.6.

The above algorithm solves the pattern occurrence detection problem in time $O(\log m)$ using $kl/\log m$ processors on a CRCW PRAM.

3.7. On-line string matching. In this section we consider the parallel version of the on-line string matching problem.

On-line Input:. A pattern string P and a sequence of l *text substrings* $T_1, T_2, \ldots, T_l$ given dynamically.

On-line Output:. For each i, $i = 1, 2, \ldots, l$, after producing the output for $T_1, T_2, \ldots, T_{i-1}$, produce the output consisting of all those positions in which P matched $T_1T_2 \ldots T_{i-1}T_i$, and that were not reported previously.

To specify the complexity of an on-line parallel algorithm, we will need to introduce a few notions. Let m denote the length of P and for each i, let k_i denote the length of T_i and let $n_i = \sum_{j=1}^{i} k_j$. Let now i be in $\{1, 2, \ldots, l\}$. Consider the time instant when the algorithm finished processing the text substrings $T_1, T_2, \ldots, T_{i-1}$ and produced the corresponding output. It is now given T_i and produces the "incremental" output. Let $T(n_{i-1}, m, k_i)$ and $P(n_{i-1}, m, k_i)$ respectively denote the time and the number of processors required by the algorithm to produce that output. The total *amortized work* done by the algorithm is defined as $\sum_{i=1}^{l} P(n_{i-1}, m, k_i)T(n_{i-1}, m, k_i)$. We say that this algorithm has *optimal amortized speedup* provided for all possible inputs the amortized work done is within a constant factor away from the time complexity of solving this problem by the best known sequential algorithm.

We will now sketch an optimal speedup (on-line) algorithm briefly. The details are easy to fill out. To simplify the exposition, consider just the case when pattern matching was done for some pattern and some string, and then the string is further extended and pattern matching needs to be done for the new longer string. Assume then, that using our algorithm from Section 2 pattern matching has been solved for pattern of length m and text string S of some length n and then an additional text string S' of length k is presented. The main idea behind the on-line implementation is to use the algorithm and data structures used in the off-line case described in Section 2, but handling each extra text chunk as it is given to the to the algorithm dynamically. Of course, it is important to be able to do this *without* recomputing most of the information that was done earlier on. The work bounds are estimated by amortizing on text chunks that are multiples of $\log m$.

We must now solve the pattern matching problem for the augmented string SS'. In general, assuming that $n > 2m$, we can write: $S = S_1S_2S_3$ where $length(S_1)$ is a positive multiple of m, $length(S_2)$ equals m, and $length(S_3)$ is smaller than m. We have all the suffix information for S_2 and all the prefix information for S_3.

We consider two cases:

1. $length(S_3) + k < m$: To avoid simple case analysis, assume $k \geq \log m$. We need to extend the prefix information available for S_3 to get the prefix information for S_3S'. By applying the automaton M, it is possible to do so in work of $O(k)$ in time of $O(\log m)$. (By refining

our algorithm, the total time could be lower for certain cases where k is smaller than m, but it is not worth considering this here.)

2. $length(S_3) + k \geq m$: Write $S' = S_a S_b$, where $length(S_a) = m - length(S_3)$. We will need to compute
   (a) The prefix information for $S_3 S_a$. This is done as described above.
   (b) The suffix information for $S_3 S_a$. This is done as in our algorithm for regular string matching, in time $O(\log m)$ and work $O(m)$.
   (c) The prefix information for S_b. This is also done as described above.

THEOREM 3.7.

The above algorithm solved the on-line pattern occurrence problem in parallel amortized linear work. The time for processing each substring is $O(\log m)$.

4. Conclusions.

In this paper, we employ s-p matching as the core computation in several pattern and string matching problems. Our main result is a parallel algorithm for computing s-p matching, which has optimal speedup on a CRCW PRAM. This algorithm is based on novel techniques that combine notions of characteristic functions, with the well-known automaton based approach to string matching that uses failure functions. Briefly, we first break the text and pattern (both of length m) appropriately into "small pieces" of size $O(\log m)$. Then, using a parallel variant of the algorithm due to Aho and Corasick [AC75] for short ($\log m$ length) strings, we group these small pieces into equivalence classes based on string equality. Given such equivalence classes, we assemble these small pieces together and solve the problem on the entire input. This is done by successively and consistently refining the equivalence classes. Using this algorithm for s-p matching as the basic building block, we specify optimal speedup parallel algorithms for several pattern and string matching problems.

REFERENCES

[ABF93] A. Amir, G. Benson and M. Farach, "Optimal Parallel Two Dimensional Pattern Matching," *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, 1993, 79-85.

[AC75] A.V. Aho and M.J. Corasick, "Efficient String Matching," *CACM*, Vol. 18, No. 6, 1975, 333-340.

[AILSV88] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin, "Parallel Construction of a Suffix Tree with applications," *Algorithmica*, Vol. 3, 1988, 347-365.

[AL88] A. Amir and G.M. Landau, "Fast Parallel and Serial Multi Dimensional Approximate Array Matching," *Theoretical Computer Science,* Vol.81, No.1, 1991, 97-115.

[Ba78] T.P. Baker, "A Technique For Extending Rapid Exact-Match String Matching to Arrays of More Than One Dimension," *SIAM J. Comput.*, Vol. 7, No. 4, 1978, 533-541.

[Bi77] R.S. Bird, "Two Dimensional Pattern Matching," *Information Processing Letters*, Vol. 6, No. 5, 1977, 168-170.

[Br74] R.P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *JACM*, Vol. 21, No. 2, 1974, 201-206.

[BG90] D. Breslauer and Z. Galil, "An optimal $O(\log \log n)$ time parallel string matching algorithm," *SIAM J. Comput.*, Vol. 19, 1990, 1051-1058.

[BJLTY91] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yanakakis, "Linear approximation of shortest superstrings," *Proc. of the 23rd ACM Symp. on Theory of Computing,* May 1991, 328-336.

[BM77] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm," *CACM*, Vol. 20, 1977, 762-772.

[CCG+93] R. Cole, M. Crochemore, Z. Galil, L. Gasieniec, R. Hariharan, S. Muthukrishnan, K. Park, and W. Rytter. "Optimally Fast Parallel Algorithms for Preprocessing and Pattern Matching in One and Two Dimensions", *Proc. of the 34th Ann. IEEE Conf. on Foundations of Computer Science*, November 1993, 248-258.

[CD88] J. L. Cornette and C. Delisi, "Some Mathematical Aspects of Mapping DNA Cosmids," *Cell Biophysics*, Vol. 12, 1988, 271-293.

[FL80] M. J. Fischer and L. Ladner, "Parallel Prefix Computation," *J. ACM,* Vol. 27, No. 4, 1980, 831-838.

[G84] Z. Galil, "Optimal Parallel Algorithms for String Matching," *Information and Control*, Vol. 67, 1985, 144-157.

[G92] Z. Galil, "A Constant-Time Optimal Parallel String-Matching Algorithm," *Proc. 23rd ACM Symposium on Theory of Computing*, 1992, pp. 69-76.

[GS83] Z. Galil and J. I. Seiferas, "Time-space Optimal String Matching," *J. Computer and Systems Sciences*, Vol. 26, 1983, 280-294. 338-355.

[H88] T. Hagerup, "On saving space in parallel computation," *Information Processing Letters*, Vol. 29, 1988, 327-329.

[J92] Joseph Ja'Ja', *An introduction to Parallel Algorithms,* Addison-Wesley Publishing Company, Inc., 1992.

[KM95] J. Kececioglu and E. Myers, "Exact and Approximate Algorithms for the Sequence Reconstruction Problem," *Algorithmica* Vol. 13, No. 1-2, 1995, in press.

[KMP77] D.E. Knuth, J.H. Morris and V.R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, Vol. 6, 1977, 323-350.

[KMR72] R.M. Karp, R.E. Miller and A.L. Rosenberg "Rapid Identification of Repeated Patterns in Strings, Trees and Arrays," *Proc. 4th ACM Symposium on Theory of Computing*, 1972, 125-136.

[KP92] Z. M. Kedem, and K. V. Palem, "Optimal Parallel Algorithms for Forest and Term matching," *Theoretical Computer Science,* Vol. 93, 1992, 245-264.

[KR87] R.M. Karp and M.O. Rabin, "Efficient Randomized Pattern-matching Algorithms," *IBM Journal of Research and Development*, Vol. 31, No. 2, March 1987, 249-260.

[M88] T.R. Mathies, "A Fast Parallel Algorithm to Determine Edit Distance," April 1988, CMU-CS-88-130.

[TU88] J. Tarhio and E. Ukkonen, "A Greedy Approximation Algorithm for Constructing Shortest Common Superstrings," *Theoretical Computer Science*, Vol. 57, 1988, 131-145.

[Tur89] J. Turner, "Approximation algorithms for the shortest common superstring problem," *Information and Computation*, 1989, Vol. 83, No. 1, 1-20.

[Ukk90] E. Ukkonen. "A linear time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 1990, Vol. 5, 313-323.

[V85] U. Vishkin, "Optimal Parallel Pattern Matching in Strings," *Information and Control*, Vol. 67, 1985, 91-113.

[V91] U. Vishkin, "Deterministic Sampling - A new Technique for Fast Pattern Matching," *SIAM J. Comput.*, Vol. 20, 1992, 22-40.

[W73] P. Weiner, "Linear Pattern Matching Algorithm," *Proc. 14 IEEE Symposium on Switching and Automata Theory,* 1973, pp1-11.