30 Parallel Algorithms on Strings

Wojciech Rytter
Warsaw University

30.1 Introduction

We present several *polylogarithmic*-time parallel string algorithms using a high level description of the *parallel random access machine* (PRAM). We select only basic problems: *string matching*, construction of the *dictionary of basic subwords*, *suffix arrays*, and *suffix trees*. Especially, the suffix tree is a very useful data structure in string algorithmics, once it is constructed it can be used to solve easily in parallel many other problems expressed in terms of suffix trees. Most problems related to trees are easily computable in parallel.

A very general model is assumed, since we are interested mainly in exposing the parallel nature of some problems without going into the details of the parallel hardware. The PRAM, a parallel version of the random access machine, is used as a standard model for presentation of parallel algorithms. The PRAM consists of a number of processors working synchronously and communicating through a common random access memory. Each processor is a random access machine with the usual operations. The processors are indexed by consecutive natural numbers, and synchronously execute the same central program; but, the action of a given processor also depends on its number (known to the processor). In one step, a processor can access one memory location. The models differ with respect to simultaneous access to the same memory location by more than one processor. For the CREW (concurrent read, exclusive write) variety of PRAM machine, any number of processors can read from the same memory location simultaneously, but write conflicts are not allowed: no two processors can attempt to write simultaneously into the same location. If we allow many processors to write in a single step into a same location, provided that they all write the same, then such a model is called CRCW PRAM. In this chapter we assume the

CREW model of the PRAM. Parallelism will be expressed by the following type of parallel statement: **for all** $i \in X$ **do in parallel** action(i). The execution of this statement consists in

- Assigning a processor to each element of X
- Executing in parallel by assigned processors the operations specified by action(i)

Usually the part "$x \in X$" looks like "$1 \leq i \leq n$" if X is an interval of integers.

A typical basic problem computed on PRAM is known as *prefix computation*. Given a vector x of n values the problem is to compute all prefix products:

$$y[1] = x[1], \quad y[2] = x[1] \otimes x[2], \quad y[3] = x[1] \otimes x[2] \otimes x[3], \quad \ldots .$$

The prefix computation problem can be computed in logarithmic time with $O(n/\log(n))$ processors.

The total work, measured as the product of time and number of processors is linear. Such parallel algorithms are called *optimal*.

There is another measure of optimality, the total number of operations. Some processors are idle at some stages and they do not contribute in this stages to the total number of operations. There is a general result that *translates* the number of operations into the total work.

Lemma 30.1 [**Brent's Lemma**], *Assume we have a PRAM algorithm working in parallel time $t(n)$ and performing $W(n)$ operations, assume also that we can easily identify processors that are not active in a given parallel step. Then we can simulate this algorithm in $O(t(n) + \lceil W(n)/t(n) \rceil)$ parallel time with $O(W(n)/t(n))$ processors.*

The lemma essentially says that in most situations the total work and total number of operations are asymptotically equal. This happens in situations that appear in this chapter.

30.2 Parallel String Matching

We say that a word is *basic* iff its length is a power of two. The **string-matching** problem is to find all occurrences of a pattern-string $x[1..m]$ in a text y of size n, where $m \leq n$.

Denote by $period(x)$ the size of the smallest period of text x, for example,

$$period(abaababa) \ = \ 5.$$

A string x is called iff $period(x) \leq |x|/2$. Otherwise it is called *aperiodic*. We consider three cases:

- **Periodic patterns**: $period(x) \leq |x|/2$
- **Aperiodic patterns**: $period(x) > |x|/2$
- **Strongly aperiodic patterns**: all basic prefixes of x are aperiodic

30.2.1 Reduction of Periodic Case to Aperiodic

Assume the pattern is periodic. Suppose that v is the shortest prefix of the pattern that is a period of the pattern. Then vv^- is called the *nonperiodic part of the pattern* (v^- denotes the word v with the last symbol removed). We omit the proof of the following lemma, which justifies the name "nonperiodic part" of the pattern.

Lemma 30.2 *If the pattern is periodic then its nonperiodic part is nonperiodic.*

Lemma 30.3 *Assume that the pattern is periodic, and that all occurrences (in the text) of its nonperiodic part are known. Then, we can find all occurrences of the whole pattern in the text in $O(\log m)$ time with $n/\log m$ processors.*

Proof. We reduce the general problem to unary string matching. Let $w = vv^-$ be the nonperiodic part of the pattern. Assume that w starts at position i on the text. By a segment containing position i we mean the largest segment of the text containing position i and having a period of size $|v|$. We assign a processor to each position. All these processors simultaneously write 1 into their positions if the symbol at distance $|v|$ to the left contains the same symbol. The last position containing 1 to the right of i (all positions between them also contain ones) is the end of the segment containing i. Similarly, we can compute the first position of the segment containing i. It is easy to compute it optimally for all positions i in $O(\log m)$ time by a parallel prefix computation. The string matching is now reduced to a search for a unary pattern, which is an easy task. ■

30.2.2 A Simple Algorithm for Strongly Aperiodic Case

Assume the pattern x is strongly aperiodic and assume that its length is a power of two (otherwise a straightforward modification is needed). Let $TestPartial(i, r)$ be the test for a match of $x[1..r]$ starting at position i in text y. The value of $TestPartial(i, r)$ is *true* iff $y[i..i + r - 1] = x[1..r]$. It can be checked in $O(\log r)$ time with $O(r/\log r)$ processors.

The algorithm is using an *elimination strategy*.

Figure 30.1 illustrates performance of this algorithm for

$$x = abaababa, \quad y = abaababaabaababa.$$

Initially, all positions are candidates for a match.

Then, we perform the following algorithm:

Algorithm *Strongly Aperiodic String Matching*
for $k = 0$ to $\log |x|$ **do**
for each candidate position i **do in parallel**
if $TestPartial(i, 2^k)$=false **then**
remove i as a candidate for a match;
return the set of remaining candidates as occurrences of x.

The algorithm *Strongly Aperiodic String Matching* works in $O(\log^2 n)$ time with $O(n)$ processors. The total number of operations is $O(n \log n)$, so the number of processors can be reduced to $O(n/\log n)$ owing to Brent's Lemma.

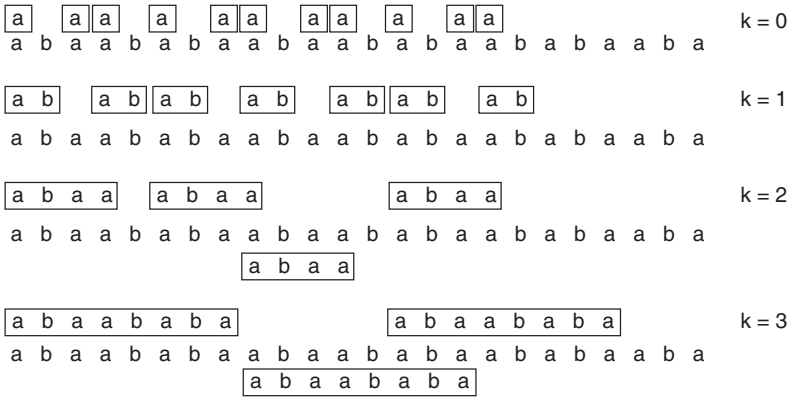

FIGURE 30.1 Performance of the algorithm *Strongly Aperiodic String Matching* for the strongly aperiodic pattern $x = abaababa$.

30.2.3 Nonperiodic Patterns: Witnesses and Duels

The first *optimal* polylogarithmic time parallel algorithm for string matching has been given by Uzi Vishkin, who introduced the crucial notion of a duel, an operation that eliminates in a constant time one of two candidates for a match. The total work of Vishkin's algorithm is $O(n)$.

We assume later in this section that the pattern x is aperiodic, which means that its smallest period is larger than $|x|/2$.

This assumption implies that two consecutive occurrences of the pattern in a text (if any) are at a distance greater than $|x|/2$. However, it is not clear how to use this property for searching the pattern. We proceed as follows: after a suitable preprocessing phase, given too close positions in the text, we eliminate one of them as a candidate for a match. In a window of size $m/2$ of the text we can eliminate all but one candidate using an elimination strategy, the time is logarithmic and the work is linear with respect to m. Hence, the total work is linear with respect to n, since there are only $O(n/m)$.

We define the following *witness table WIT*: for $0 < i < |m|$,

$$WIT[i] = \textit{any } k \text{ such that } x[i+k-1] \neq x[k], \quad \text{or}$$
$$WIT[i] = 0, \quad \text{if there is no such } k.$$

This definition is illustrated in Figures 30.2 and 30.3.

A position i_1 on y is said to be *in the range of* a position i_2 if $|i_1 - i_2| < m$. We also say that two positions $i_1 < i_2$ on the text are *consistent* if i_2 is not in the range of i_1, or if $WIT[i_2 - i_1] = 0$.

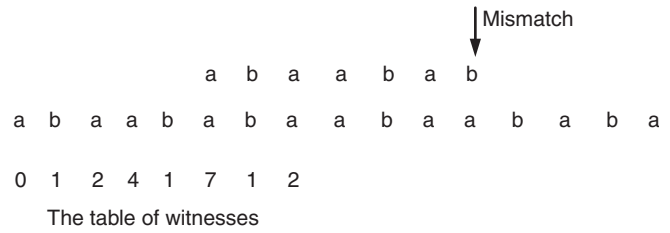

FIGURE 30.2 The witness table $WIT = [0, 1, 2, 4, 7, 1, 2]$ for the text *abaababaabaababa*, $WIT[7] = 7$, owing to the mismatch shown in the figure by the arrow.

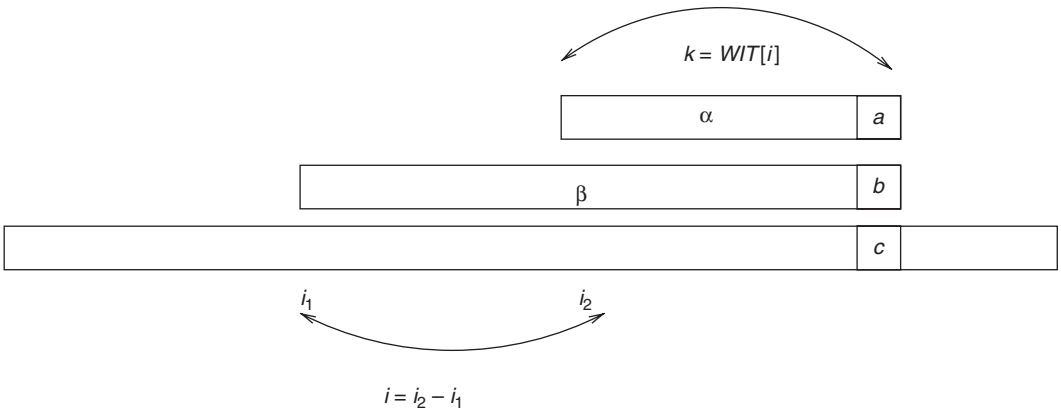

FIGURE 30.3 The duel between positions $i1$, $i2$ and computation of the witness for the loser. We know that $a \neq b$, so $a \neq c$ or $b \neq c$. The strings αa, βb are prefixes of the pattern. If $i2$ is a loser then we know a possible value $WIT[i2] = k$, otherwise a possible value $WIT[i1] = i + k - 1$ is known.

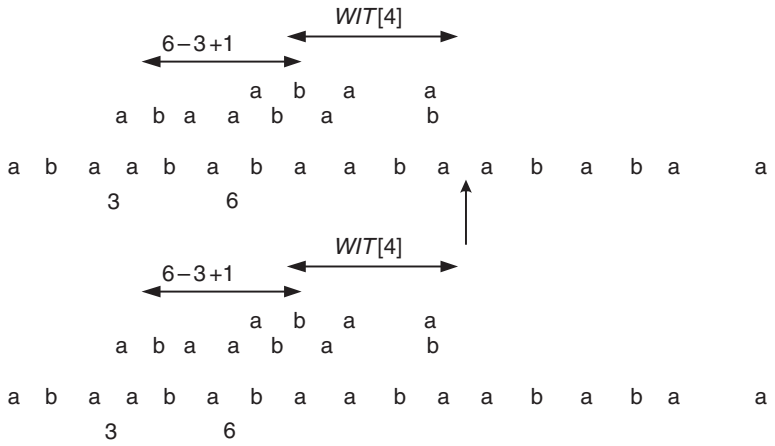

FIGURE 30.4 A duel between two inconsistent positions 3 and 6. The first of them is eliminated since the prefix of the pattern of size 7 placed at starting position 3 does not match the pattern at position $6 + WIT[6 - 3 + 1] - 1$. By the way, we know that possible value of $WIT[3]$ is 7.

If the positions are not consistent, then we can remove one of them as a candidate for the starting position of an occurrence of the pattern just by considering position $i_2 + WIT[i_2 - i_1 + 1]$ on the text. This is the operation called a **duel** (see Figure 30.4).

Let $i = i_2 - i_1 + 1$, and $k = WIT[i]$. Assume that we have $k > 0$, that is, positions i_1, i_2 are not consistent. Let $a = x[k]$ and $b = x[i + k - 1]$, then $a \neq b$. Let c be the symbol in the text y at position $i_2 + k - 1$. We can eliminate at least one of the positions i_1 or i_2 as a candidate for a match by comparing c with a and b.

In some situations, both positions can be eliminated, but, for simplicity, the algorithm below always removes exactly one position.

Let us define, with $a = x[WIT[i_2 - i_1 + 1]]$:

$$duel(i_1, i_2) = (\textbf{if } a = c \textbf{ then } i_1 \textbf{ else } i_1).$$

The position that "survives" is the value of *duel*, the other position is eliminated. We also say that the survival position is a **winner** and the other one is a **loser**.

The *duel* operation is illustrated in Figures 30.4 and 30.3.

In the preprocessing phase we have $y = x$, the pattern makes duels on itself.

30.2.4 Preprocessing the Pattern

In the preprocessing phase we assume that $y = x$ and each duel makes additional operation of computing the witness value for the losing position. This means that if $k = duel(i_1, i_2)$ is the *winning position* and $j = \{i_1, i_2\} - \{k\}$, then j is a *loser* and as a side-effect of $duel(i_1, i_2)$ the value of $WIT[j]$ is computed.

We use a straightforward function $ComputeWit(i)$ which computes the first witness j for i in time $O(\log n \cdot \log\lceil (i+j)/i \rceil)$ using $O(i+j)$ processors. If there is no witness then define $ComputeWit(i) = 0$. The function uses a kind of prefix sum computation to find the first mismatch for $j \in [1 \ldots i]$, then for $j \in [1 \ldots 2i]$ and so forth, doubling the "distance" until the first witness is computed or the end of x is reached.

The preprocessing algorithm operates on a logarithmic sequence of *prefix windows* that has sizes growing at least as powers of two.

The i-th **window** $\mathcal{W}_i$ is an interval $[1 \ldots j]$. The algorithm maintains the following **invariant**:

after the i-th iteration the witnesses for positions in W_i are computed and for each $k \in \mathcal{W}_i$, $WIT[k] \in W_i$ or x is periodic with period k,
the size of the window $\mathcal{W}_i$ is at least 2^i.

If $\mathcal{W}_i$ is an interval $[1 \ldots j]$ then define the i-th **working interval** as $\mathcal{I}_i = [j+1 \ldots 2j]$.

The preprocessing algorithm works in iterations, in the $(i+1)$-st **iteration** we perform as follows: witnesses are computed for the positions in the *working interval* $\mathcal{I}_i$ using already computed part of the witness table for positions in $\mathcal{W}_i$ and performing in parallel the tree of duels; see Figures 30.5 and 30.6. The witnesses for losers are computed. Only one position j in $[k+1 \ldots 2k]$, the final winner, has no witness value. We compute its witness value k in parallel using *ComputeWit*(j). There are three cases:

Case 1: aperiodic case: $k \in \mathcal{W}_i$, $x[1 \ldots j-1]$ does not start a periodic prefix, see Figure 30.6. The iteration is finished. New window $calW_{i+1}$ is twice larger than $\mathcal{W}_i$.

Case 2: periodic case: $k > 0$ and $k \notin \mathcal{W}_i$, $x[1 \ldots j-1]$ starts a periodic prefix that ends at k. Then the witness table for positions in $[j \ldots k-j$ are computed using periodicity copying from the witness values in $\mathcal{W}_i$, see Figure 30.7.

Case 3: $k = 0$, the whole pattern is periodic. We have already computed witness table for aperiodic part. This is sufficient for the searching phase, owing to Lemma 30.3.

Our construction implies the following fact.

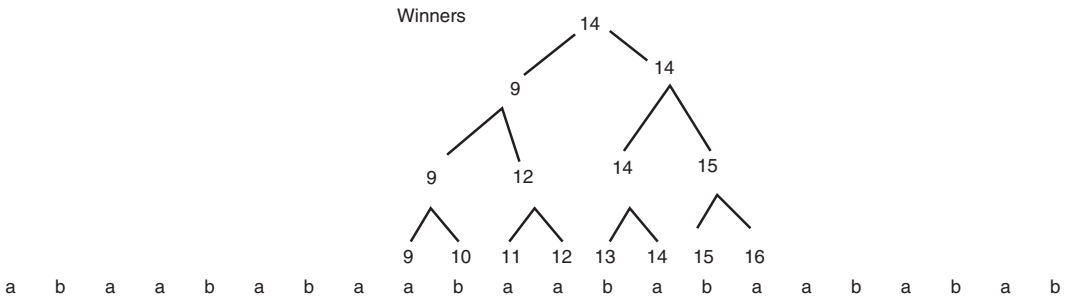

FIGURE 30.5 Preprocessing the pattern *abaababaabaababaababab* . . ., the iteration corresponds to the *working interval* in x of positions [9..16]: the tree of winners in the duels between positions in the interval [9..16].

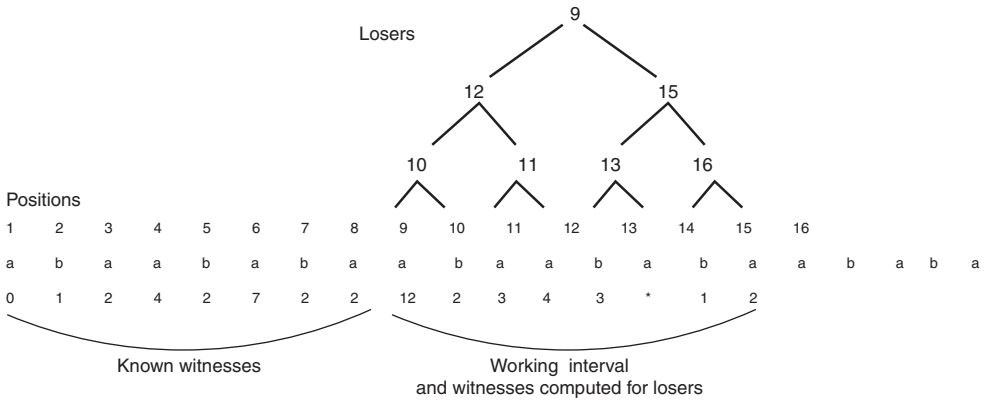

FIGURE 30.6 Processing the working interval [9..16] and computing WIT[9..16]: the tree of losers in the duels between positions in the interval [9..16]. For each loser, as a side-effect of the corresponding duel, the value of its witness is computed. The witness of position 14 is not computed when making duels since it has not lost any duel and it should be computed by applying the procedure *ComputeWit*(14). It computes $WIT[14] = 9$.

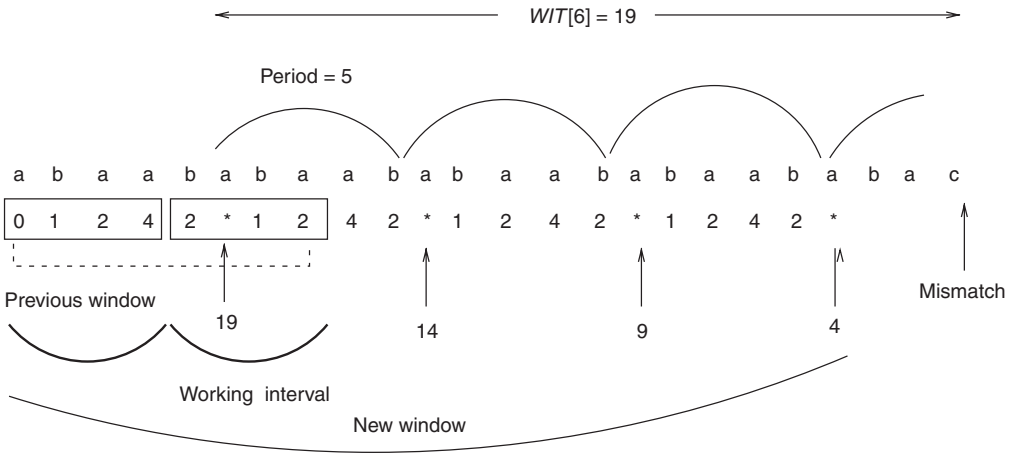

FIGURE 30.7 Periodic case, one iteration of preprocessing the pattern $(abaab)^4 aba\#$. During computation of witnesses in the working interval [5..8] we found that the witness for position 6 is very far away: $ComputeWit(6) = WIT[6] = 19$. This implies a long periodic segment and witnesses are copied to positions to the right of the working interval in a periodic way, except positions 6, 6+5, 6+10, 6+15 denoted by $*$. For these positions, the witnesses are computed by the formula $WIT[6 + 5k] = 19 - 5k$ for $k = 0, 1, 2, 3$.

Lemma 30.4 *We can locate nonperiodic part of x (possibly the whole x) and compute its witness table in $O(\log^2 n)$ parallel time with $O(n)$ totalwork.*

30.2.5 Searching Phase

Assume that the pattern is aperiodic and the witness table has been computed. Define the operation $\otimes$ by

$$i \otimes j = duel(i, j).$$

The operation $\otimes$ is "practically" associative. This means that the value of $i_1 \otimes i_2 \otimes i_3 \otimes \ldots \otimes i_{m/2}$ depends on the order of multiplications, but all values (for all possible orders) are good for our purpose. We need any of the possible values.

Once the witness table is computed, the string-matching problem reduces to instances of the parallel prefix computation problem. We have the following algorithm. Its behavior on an example string is shown in Figure 30.8 for searching the pattern *abaababa*, for which the witness table is precomputed:

Algorithm *Vishkin-string-matching-by-duels*;
consider windows of size $m/2$ on y;
{ sieve phase }
for each window **do in parallel**
{ $\otimes$ can be treated as if it were associative }
compute the surviving position $i_1 \otimes i_2 \otimes i_3 \otimes \cdots \otimes i_{m/2}$
where $i_1, i_2, i_3, \ldots, i_{m/2}$ are consecutive positions
in the window;
{ naive phase }
for each surviving position i **do in parallel**
check naively an occurrence of x at position i
using m processors;

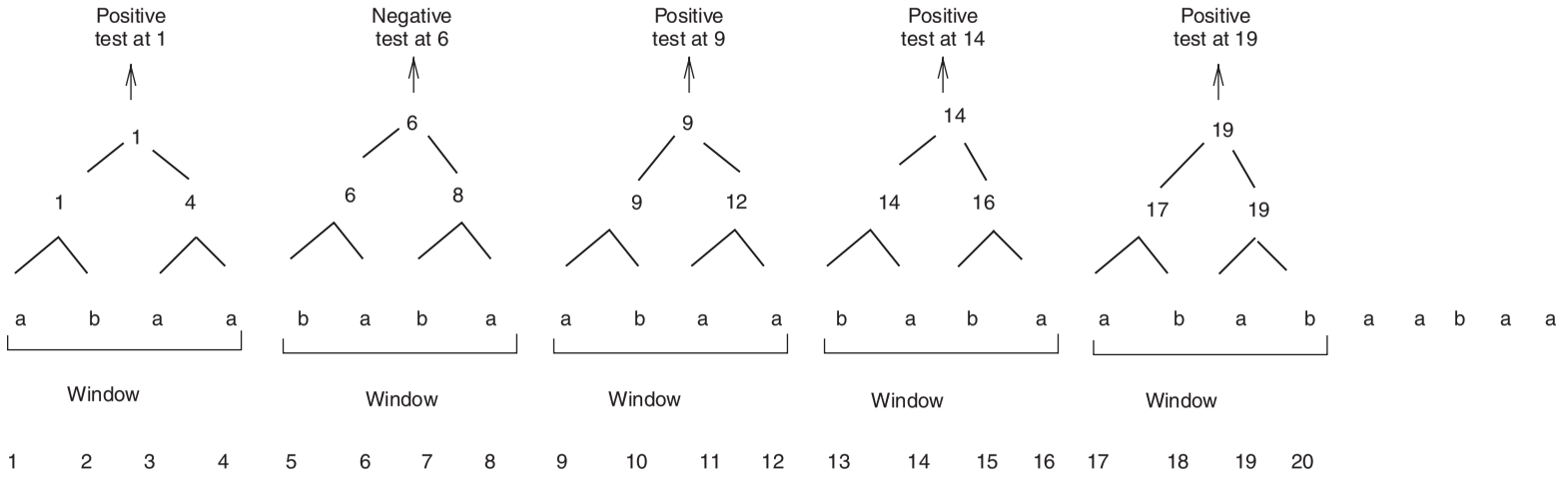

FIGURE 30.8 Searching for *abaababa* in *abaababaabaababaababaabaa*, the witness values *WIT*[1..4] for the first *half* of the pattern are used to eliminate in each window all candidates but one, for a possible match of *abaababa*.

Theorem 30.1 *Assume we know the witness table and the period of the pattern. Then, the string-matching problem can be solved optimally in $O(\log m)$ time with $O(n/\log m)$ processors of a CREW PRAM.*

Proof. Let $i_1, i_2, i_3, \ldots, i_{m/2}$ be the sequence of positions in a given window. We can compute $i_1 \otimes i_2 \otimes i_3 \otimes \cdots \otimes i_{m/2}$ using an optimal parallel algorithm for the parallel prefix computation. Then, in a given window, only one position survives; this position is the value of $i_1 \otimes i_2 \otimes i_3 \otimes \cdots \otimes i_{m/2}$. This operation can be executed simultaneously for all windows of size $m/2$.

For all windows, this takes $O(\log m)$ time with $O(n/\log m)$ processors of a CREW PRAM.

Afterward, we have $O(n/m)$ surviving positions altogether. For each of them we can check the match using $m/\log m$ processors. Again, a parallel prefix computation is used to collect the result, that is, to compute the conjunction of m Boolean values (match or mismatch, for a given position). This takes again $O(\log m)$ time with $O(n/\log m)$ processors.

Finally, we collect the $O(n/m)$ Boolean values using a similar process. ■

Corollary 30.1 *There is an $O(\log^2 n)$ time parallel algorithm that solves the string-matching problem (including preprocessing) with $O(n/\log^2 n)$ processors of a CREW PRAM.*

30.3 Naming Technique

The numbering or naming of subwords of a text, corresponding to the sorted order of these subwords, can be used to build a useful data structure. We assign names to certain subwords, or pairs of subwords. Assume we have a sequence

$$S = (s_1, s_2, \ldots, s_t)$$

of at most n different objects. The *naming* of S is a table

$$X[1], X[2], \ldots, X[t]$$

that satisfies conditions (1–2). If, in addition, it satisfies the third condition then the naming is called a *sorted naming*.

1. $s_i = s_j \Leftrightarrow X[i] = X[j]$, for $1 \leq i, j \leq t$
2. $X[i] \in [1..n]$ for each position i, $1 \leq i \leq t$
3. $X[i]$ is the rank of s_i in the ordered list of the different elements of S

Given the string y of length n, we say that two positions are k-equivalent if the subwords of length k starting at these positions are equal. Such an equivalence is best represented by assigning to each position i a name or a number to the subword of length k starting at this position. The name is denoted by $Name_k[i]$ and called a k-name. We assume that the table $Name_k$ is a good and sorted numbering of all subwords of a given length k.

We consider only those subwords of the text whose length k is a power of two. Such subwords are called *basic subwords*. The *name of a subword* is denoted by its rank in the lexicographic ordering of subwords of a given length. For each k-name r we also require (for further applications) a link $Pos_k[r]$ to any one position at which an occurrence of the k-name r starts. Symbol # is a special end marker that has the highest rank in the alphabet. The text is padded with enough end markers to let the k-name at position n defined.

The tables *Name* and *Pos* for a given text w are together called its **dictionary of basic subwords** and is denoted by $DBS(w)$.

Example The tables below show the dictionary of basic subwords for an example text: tables of k-names and of their positions. The k-name at position i corresponds to the subword $y[i..i + k - 1]$; its name is

its rank according to lexicographic order of all subwords of length k (order of symbols is $a < b < \#$). Indices k are powers of two. The tables can be stored in $O(n \log n)$ space.

Positions	=	1	2	3	4	5	6	7	8			
y	=	a	b	a	a	b	b	a	a	#	#	#
$Name_1$	=	1	2	1	1	2	2	1	1			
$Name_2$	=	2	4	1	2	5	4	1	3			
$Name_4$	=	3	6	1	4	8	7	2	5			
Name of subword	=	1	2	3	4	5	6	7	8			
Pos_1	=	1	2									
Pos_2	=	3	1	8	2	5						
Pos_4	=	3	7	1	4	8	2	6	5			

Remark String matching for y and pattern x of length m can be easily reduced to the computation of a table $Name_m$. Consider the string $w = x\&y$, where & is a special symbol not in the alphabet. Let $Name_m$ be the array which is part of $DBS(x\&y)$. If $q = Name_m[1]$ then the pattern x starts at all positions i on y such that $Name_m[i + m + 1] = q$.

The tables above display $DBS(abaabbaa\#)$. Three additional #'s are appended to guarantee that all subwords of length 4 starting at positions 1, 2, …, 8 are well defined. The figure presents tables *Name* and *Pos*. In particular, the entries of Pos_4 give the lexicographically sorted sequence of subwords of length 4. This is the sequence of subwords of length 4 starting at positions 3, 7, 1, 4, 8, 2, 6, 5. The ordered sequence is

$$aabb,\ aa\#\#,\ abaa,\ abba,\ a\#\#\#,\ baab,\ baa\#,\ bbaa.$$

We introduce the procedure *Sort-Rename* that is defined now. Let S be a sequence of total size $t \leq n$ containing elements of some linearly ordered set. The output of *Sort-Rename*(S) is an array of size t, which is a good and sorted naming of S.

Example Let $S = (ab,\ aa,\ ab, ba, ab, ba, aa)$. Then,

$$\textit{Sort-Rename}(S) = (2,\ 1,\ 2,\ 3,\ 2,\ 3,\ 1).$$

For a given k, define

$$\textit{Composite-Name}_k[i] = (Name_k[i], Name_k[i + k]).$$

KMR algorithm is based on the following simple property of naming tables.

Lemma 30.5 [**Key-Lemma**], $Name_{2k} = \textit{Sort-Rename}(\textit{Composite-Name}_k)$.

The main part of algorithm *Sort-Rename* is the lexicographic sort. We explain the action of *Sort-Rename* on the following example:

$$x = ((1, 2), (3, 1), (2, 2), (1, 1), (2, 3), (1, 2)).$$

The method to compute a vector X of names satisfying conditions (1–3) is as follows. We first create the vector y of composite entries $y[i] = (x[i], i)$. Then, the entries of y are lexicographically sorted. Therefore, we get the ordered sequence

$$((1, 1), 4), ((1, 2), 1), ((1, 2), 6), ((2, 2), 3), ((2, 3), 5), ((3, 1), 2).$$

Next, we partition this sequence into groups of elements having the same first component. These groups are consecutively numbered starting from 1. The last component i of each element is used to define the output element $X[i]$ as the number associated with the group. Therefore, in the example, we get

$$X[4] = 1,\ X[1] = 2,\ X[6] = 2,\ X[3] = 3,\ X[5] = 4,\ X[2] = 5.$$

Doing so, the procedure *Sort-Rename* has the same complexity as sorting n elements.

Lemma 30.6 *If the vector x has size n, and its components are pairs of integers in the range $(1, 2, \ldots, n)$, Sort-Rename(x) can be computed in parallel time $O(\log n)$ with $O(n)$ processors.*

The dictionary of basic subwords is computed by the algorithm Compute-DBS. Its correctness results from fact $(*)$ below. The number of iterations is logarithmic, and the dominating operation is the call to procedure *Sort-Rename*.

Once all vectors $Name_p$, for all powers of two smaller than r, are computed, we easily compute the vector $Name_q$ in linear time for each integer $q < r$. Let t be the greatest power of two not greater than q. We can compute $Name_q$ by using the following fact:

$(*)$ $Name_q[i] = Name_q[j]$ iff $Name_t[i] = Name_t[j]$ and $Name_t[i + q - t] = Name_t[j + q - t]$

```
Algorithm Compute-DBS;
{ a parallel version of the Karp-Miller-Rosenberg algorithm }
    K := largest power of 2 not exceeding n;

    Name_1 := Sort-Rename(x);
    for k := 2, 4, ..., K do
        Name_2k := Sort-Rename(Composite-Name_k);

    for each k = 1, 2, 4, ..., K do in parallel
        for each 1 ≤ i ≤ n do in parallel
            Pos_k[Name_k[i]] := i;
```

This construction proves the following theorem.

Theorem 30.2 *The dictionary of basic subwords of a text of length n can be constructed in $\log^2(n)$ time with $O(n)$ processors of a CREW PRAM.*

We show a straightforward application of the dictionary of basic subwords. Denote by *LongestRepFactor*(x) the length of the *longest repeated subword* of y. It is the longest word occurring at least twice in y (occurrences are not necessarily consecutive). When there are several such longest repeated subwords, we consider any of them. Let also $LongestRepFactor_k(x)$ be the maximal length of the subword that occurs at least k times in y. Let us denote by $REP_k(r, y)$ the function that tests is there is a k-repeating subword of size r. Such function can be easily implemented to run in $\log n$ parallel time if we have $DBS(w)$.

Theorem 30.3 *The function $LongestRepFactor_k(x)$ can be computed in $O(\log^2 n)$ time with $O(n)$ processors.*

Proof. We can assume that the length n of the text is a power of two; otherwise a suitable number of "dummy" symbols are appended to the text. The algorithm *KMR* is used to compute $DBS(x)$. We

then apply a kind of binary search using function $REP_k(r, y)$: the binary search looks for the maximum r such that $REP_k(r, y) \neq$ nil. If the search is successful then we return the longest (k times) repeated subword. Otherwise, we report that there is no such repetition. The sequence of values of $REP_k(r, y)$ (for $r = 1, 2, \ldots, n-1$) is "monotonic" in the following sense: if $r1 < r2$ and $REP_k(r2, y) \neq$ nil, then $REP_k(r1, y) \neq$ nil. The binary search behaves similarly to searching an element in a monotonic sequence. It has $\log n$ stages; at each stage the value $REP_k(r, y)$ is computed in logarithmic time. Altogether the computation takes $O(\log n)$ parallel time. ■

30.4 Parallel Construction of Suffix Arrays

Assume we are given a string x of length n, assume also that we append to x a special end marker symbol $x_{n+1} = \#$, which is smaller than any other symbol of x. We assume in the paper that the alphabet is a subset of $[1 \ldots n]$.

The **suffix array** $SUF = [i_1, i_2, \ldots, i_n]$ is the permutation that gives the sorted list of all suffixes of the string x. This means that

$$suf(i_1) < suf(i_2) < suf(i_3) < \cdots < suf(i_n),$$

where $<$ means here lexicographic order, and $suf(i_k) = x[i_k \ldots n]\#$.

In other words $suf(i_k)$ is the k-th suffix of x in sense of lexicographic order (Figure 30.9).

There is another useful table LCP of the lengths of adjacent common prefixes:

For $1 < i < n$, $LCP[k]$ is the length of the longest common prefix of $suf(i_k)$ and $suf(i_{k-1})$.

In addition, define $LCP[1] = 0$.

AQ: Please confirm whether the placement of Figures 30.9 and 30.12 are appropriate.

Example Let us consider an example string: $x = b\ a\ b\ a\ a\ b\ a\ b\ a\ b\ b\ a\ \#$.

For this string we have:

$$SUF = [4,\ 2,\ 5,\ 7,\ 9,\ 12,\ 3,\ 1,\ 6,\ 8,\ 11,\ 10]$$

$$LCP = [0\ ,1,\ 3,\ 4,\ 2,\ 1,\ 0,\ 2,\ 4,\ 3,\ 2,\ 1]$$

FIGURE 30.9 The sorted sequence (top-down) of 12 suffixes of the string $x = babaababbba\#$ with the *LCP-values* between suffixes (length of shaded straps). The special suffix # and the empty suffix are not considered.

The data structure consisting of both tables SUF, LCP is called the *suffix array*. The dictionary of basic subwords can be used to show directly the following fact.

Theorem 30.4 [12] *The suffix array can be constructed in* $O(\log^2 n)$ *time with* $O(n)$ *processors.*

The work of the algorithm from this theorem is $O(n \log^2 n)$.

Once $DBS(x)$ and suffix arrays of x has been computed the table *LCP* can be computed in a straightforward way in $O(\log n)$ time with $O(n)$ processors. We perform similarly as in the computation of the longest repeating subword. We assign a single processor to each i that computes the length of the longest common prefix between two consecutive (in lexicographic order) suffixes of x in logarithmic sequential time using a kind of binary search.

The work of constructing suffix arrays has been improved in Reference 12 to $O(n \log n)$ using the *Skew Algorithm* which computed only a part of the dictionary *DBS* of basic subwords of linear size. Observe that the whole dictionary *DBS* has $O(n \log n)$ size.

Theorem 30.5 [12] *The arrays SUF, LCP can be constructed in* $O(\log^2 n)$ *time and* $O(n \log n)$ *work.*

We shortly describe the construction behind the proof of this theorem. The construction is especially interesting in sequential setting where a linear time algorithm is given.

Let $Z \subseteq \{1, 2, 3, \ldots\}$. Denote by SUF_Z the part of the suffix array corresponding only to positions in Z. The set Z is *good* iff for each x of length n it satisfied the following conditions: (1) We can reduce the computation of SUF_Z of x to the computation of *SUF* for a string x' of size at most cn, where $c < 1$ is a constant; (2) If we know the table SUF_Z then we can compare lexicographically any two suffixes of x in constant work.

We take later $Z = \{i \geq 1 : \ i \text{ modulo } 3 \in \{0, 2\}\}$. We leave to the reader the proof that our set Z satisfies the second condition for a *good* set. We show on an example how the first condition is satisfied.

Example For our example string $x = babaababababba\#$ we have

$$SUF_Z = [2, 5, 9, 12, 3, 6, 8, 11].$$

Sorting suffixes starting at positions 2, 5, 8, 11 correpsonds to sorting suffixes of the string $[aba][aba][bab][ba\#]$, where 3-letter blocks can be grouped together. Similarly sorting suffixes starting at positions 3, 6, 9, 12 corresponds to sorting suffixes of the string $[baa][bab][bab][abb][a\#\#]$. We can encode, preserving lexicographic order, the blocks into single letters A, B, C, D, E, F; see Figure 30.10, where it is shown how the computation of SUF_Z is reduced to the computation of the suffix array for $x' = BBFD\$EFCA$. The symbol \$ is lexicographically smallest.

The *Skew Algorithm* recursively computes the suffix array for x'. Then we sort all suffixes using the property (2) of *good* sets. We can reduce it to merging, since the sort of suffixes at unclassified positions (outside Z) can be done by referring to the sorting sequence of positons in Z. Then we can merge two sorted sequence, and this can be done in logarithmic time with $O(n)$ work.

The table *LCP* can be computed in the process of parallel computation of the suffix array in the *Skew Algorithm*. This means that we recursively compute the suffix array and *LCP* table for the smaller string x', then we reconstruct the suffix array together with *LCP* for the initial string x. We refer the reader to Reference 12.

30.5 Transformation of Suffix Arrays into Suffix Trees

A simple algorithm for suffix arrays suggests that the simplest way to compute in parallel suffix trees could be to transform in parallel suffix arrays into suffix trees. Assume that the tables SUF and LCP for

an input string are already computed. We show two simple transformations. Especially, the first one is very simple. If we compute suffix arrays using $DBS(x)$, then descibed below algorithm BUILD1 is a very unsophisticated construction of the suffix tree in polylogarithmic time with the work only slightly larger than for other algorithms. We have simplicity against sophisticated, losing a polylogarithmic factor.

30.5.1 Algorithm BUILD1

We use a parallel *divide-and-conquer* approach. Its correctness is based on the following fact.

Observation Assume $i < j < k$, then the lowest common ancestor in the suffix tree of leaves $SUF[i]$ and $SUF[j]$ is on the branch from the root to the leaf $SUF[k]$. Hence, for a fixed k all lowest common ancestors of $SUF[i]$ and $SUF[j]$ for all $i < k < j$ are on a same branch.

Let $SUF = [i_1, i_2, \ldots, i_n]$. The input to the function is a subsegment α of the sequence $[i_1, i_2, \ldots, i_n]$.

The suffix trees are built for the subsegments of of $suf(i_1), suf(i_2), \ldots, suf(i_n)$. α is split into two parts α_1, α_2 of approximately same size. The suffix trees $T1$ and $T2$ are constructed in parallel for the left and the right parts. The last element of α_1 is the same as of α_2. Hence the resulting suffix tree results by merging together the rightmost branch of $T1$ with the leftmost branch of $T2$. The algorithm is written as a recursive function $BUILD1$.

Example In our example, see Figure 30.11b, the sequence corresponding to the suffix array is $\alpha = [i_1, i_2, \ldots, i_n] = [12, 11, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8]$. Let us consider SUF[6] = 1, and the branch from the root to 1 is a separator of the suffix tree, it separates the tree into two subtrees with a common branch: the first subtree is for the first half $\alpha_1 = [12, 11, 3, 6, 9, 1]$ and the second subtree for $\alpha_2 = [1, 4, 7, 10, 2, 5, 8]$. The sizes of "halves" differ at most by one.

Theorem 30.6 *Assume that the arrays SUF, LCP are given for an input string of size n. Then the algorithm BUILD1 computes the suffix tree for x in $O(\log^2 n)$ time and $O(n)$ space with $O(n \log n)$ total work.*

Proof. If $|\alpha| = 2$ and we have only two suffixes $suf(i_{j-1})$, $suf(i_j)$, for some j, then we can create for them a partial suffix tree in $O(1)$ time as follows. The tree has a root, two leaves corresponding to whole suffixes $suf(i_{j-1})$, $suf(i_j)$, and one internal node corresponding to the longest common prefix of $suf(i_{j-1})$, $suf(i_j)$. The length of this prefix is given by LCP[j].

The main operation is that of *merging* two branches corresponding to a same suffix. The *total depth* of a node is the length of a string corresponding to this node in the suffix tree. The nodes on the branches

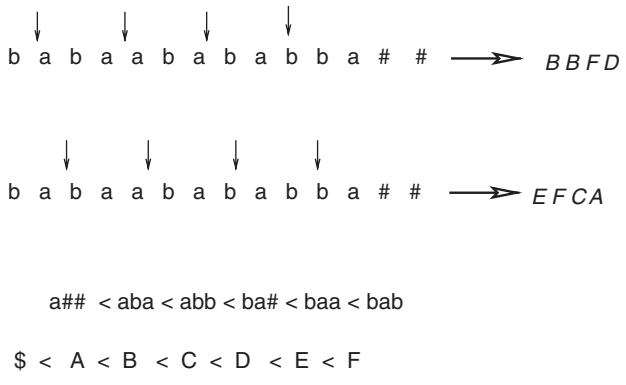

FIGURE 30.10 Sorting suffixes starting at positions pointed by arrow in the text $x = babaababababba\#$ corresponds to sorting suffixes of a "compressed" representation $x' = BBFD\$EFCA$. The set $Z \cap \{1, 2, \ldots, n\}$ corresponds to positions pointed by arrows.

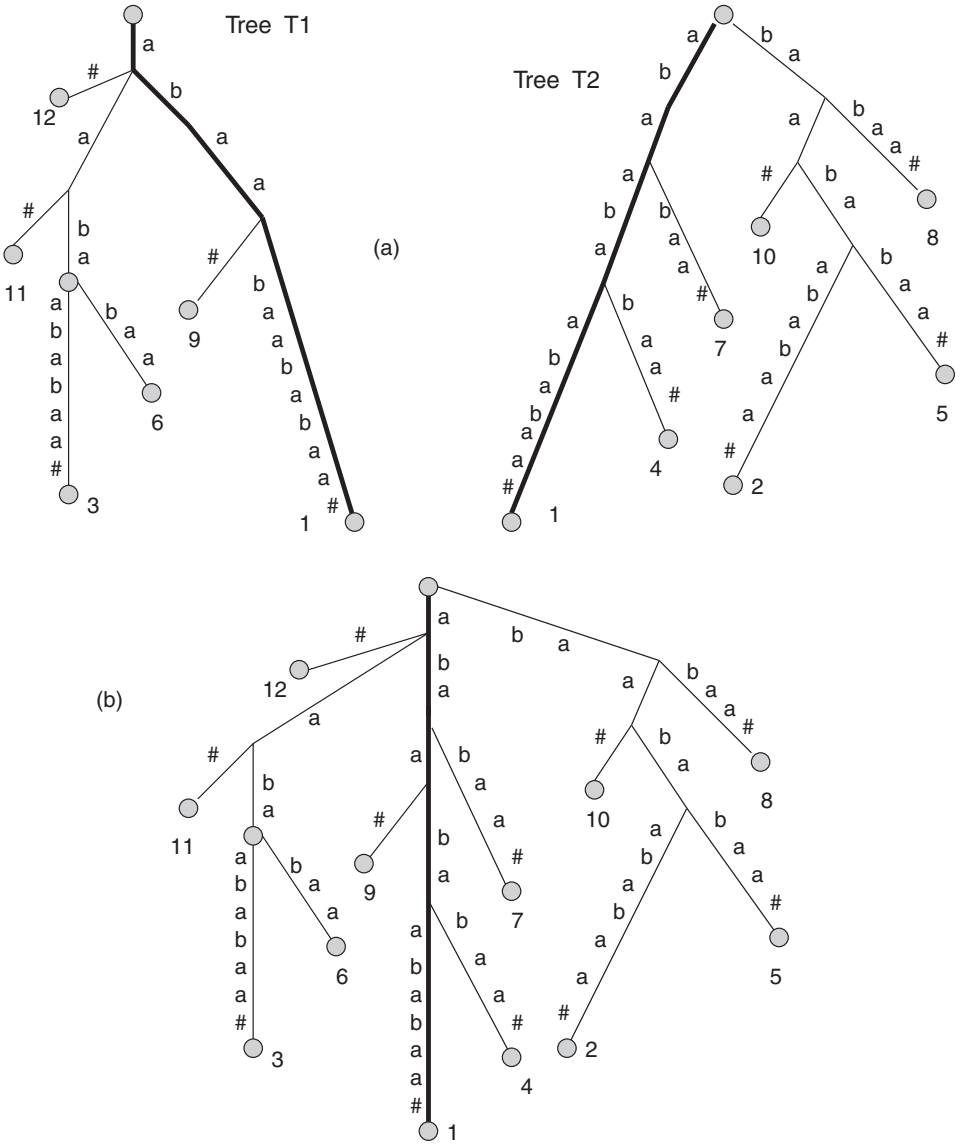

FIGURE 30.11 (a) The suffix trees for two "halves" of the sorted sequence of suffixes: (12, 11, 3, 6, 9, 1) and (1, 4, 7, 10, 2, 5, 8). The "middle" suffix (in bold) belongs to both parts. (b) The suffix tree resulting after merging the nodes corresponding to the common suffix branch corresponding to *suf*(1).

are sorted in sense of their total depth. Hence to merge two branches we can do a parallel merge of two sorted arrays of size n.

This can be done in logarithmic time with linear work, see Reference 45. The total time is $O(\log^2 n)$ since the depth of the recursion of the function $BUILD1(\alpha)$ is logarithmic. ■

The total work satisfies similar recurrence as the complexity of the merge-sort.

$$Work(n) = Work(n/2) + Work(n/2 + 1) + O(n).$$

The solution is a function of order $O(n \log n)$. This completes the proof.

Algorithm BUILD1(α); { Given SUF = $[i_1, i_2, \ldots i_n]$}
{ $\alpha = (s_1, s_2, \ldots s_k)$ is a subsegment of $[i_1, i_2, \ldots i_n]$}
if $|\alpha| \leq 2$ **then**
compute the suffix tree T sequentially in $O(1)$ time;
{Comment: the table *LCP* is needed for $O(1)$ time computation in this step}
else
in parallel do
{$T1$:= BUILD1($s_1, s_2, \ldots s_{k/2}$); $T2$:= BUILD1($s_{k/2}, s_{k/2+1}, \ldots s_k$)};
merge in parallel rightmost branch of T1 with leftmost branch of $T2$;
return the resulting suffix tree T;

The history of the algorithm for our example string is shown in Figure 30.1.

30.5.2 Algorithm BUILD2

We design now the algorithm BUILD2, the preprocessing phase for this algorithm consists in the computation of two tables: the *Nearest Smaller Neighbor* table *NS* and the table *Leftmost*$[1 \ldots n]$. We show that both tables can be computed in $O(\log n)$ time with $O(n)$ processors. For all $i > 1$ such that $LCP[i] > 0$ define:

$$SN[i] = \begin{cases} L[i], & \text{if } LCP[L[i]] \geq LCP[R[i]], \\ R[i], & \text{otherwise,} \end{cases}$$

where $L[i] = \max\{j < i\colon LCP[j] < LCP[i]\}$ $R[i] = \min\{j > i\colon LCP[j] < LCP[i]\}$.

$L[i]$ is the first position to the left of i with smaller value, symmetrically $R[i]$ is the first smaller neighbor on the right side. *Leftmost*$[i]$ is defined for each $1 \leq i \leq n$ as:

$$\min\{1 \leq j \leq i\colon\ LCP[j] = LCP[i] \text{ and } LCP[k] \geq LCP[i] \text{ for each } j \leq k \leq i\ \}.$$

In other words, *Leftmost*$[i]$ is the leftmost position to the left of i with the same value as in position i and such that intermediate values are not smaller (Figure 30.12).

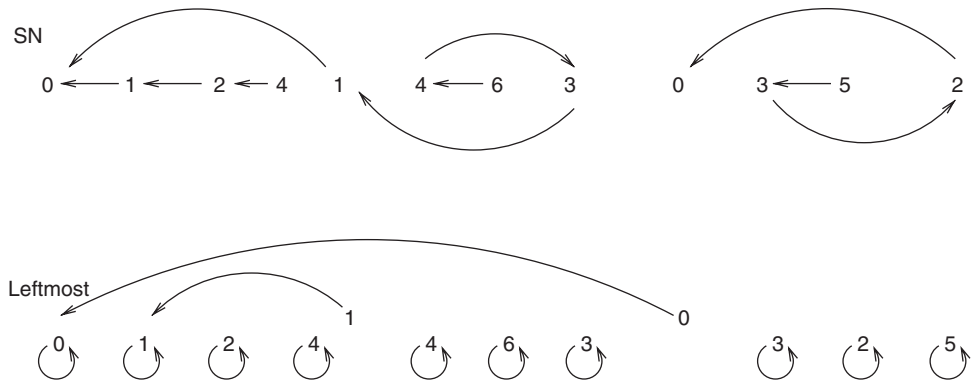

FIGURE 30.12 The tables *SN* and *Leftmost*.

Lemma 30.7 [**Preprocessing-Lemma**], *The tables SN and Leftmost can be computed on a CREW PRAM in $O(\log n)$ time with n processors.*

Proof. We show how to compute the table L within claimed complexities. Assume w.l.o.g. that n is a power of 2 and construct complete regular binary tree R of height $\log n$ over positions $1 \ldots n$. Using a classical algorithm for parallel prefix computation we compute for each node v of R the minimum of LCP over the interval corresponding to leaf-positions of the subtree rooted at v. Then for each position i we compute $L[i]$ by first traversing up in R from the i-th leaf until we hit a node v whose value is smaller then $LCP[i]$, and than we go down to the position $L[i]$. ■

The second algorithm is nonrecursive. Assume that the sequence of leaves $(i_1, i_2, \ldots, i_n)$ read from left to right corresponds to the lexicographically ordered sequence of suffixes. The leaf nodes are represented by starting positions i_k of suffixes. Denote by LCA the lowest common ancestor function.

Fact 30.7 *Each internal node v equals $LCA(i_{k-1}, i_k)$ for some k. As the representative of v we choose i_k such that in the sequence $(i_1, i_2, \ldots, i_n)$ i_k is leftmost among all ri_r such that $LCA(i_{k-1}, i_k) = v$, see Figure 30.13. The internal node $i_k = LCA(i_{k-1}, i_k)$ is the node on the branch from the root to i_k, the string corresponding to the path from root to v is of length $LCP[k]$.*

Denote by $string\_repr(v)$ the string corresponding to labels from the root to v. We have:

If i_k is an internal node then $string\_repr(i_k) = x[i_k \ldots i_k + LCP[k] - 1]$;
Ifi_k is a leaf node then $string\_repr(i_k) = x[i_k \ldots n]$.

The following fact follows directly from these properties.

Lemma 30.8 [**Edge-Labels-Lemma**], *Assume i_s is the father of i_k and $r = LCP[k]$, then the label of the edge from i_s to i_k is*
(a.) If i_k is an internal node then $x[i_k + p \ldots i_k + r - 1]$, where $p = LCP[s]$.
(b.) $x[i_k + r \ldots n]$, otherwise.

If i_k is an internal node then we denote its father by $InFather[i_k]$, otherwise we denote it by $LeafFather[i_k]$. information whether it is a leaf or internal node.

Lemma 30.9 [**Fathers-Lemma**], *Assume $(i_1, i_2, \ldots, i_n)$ is the suffix array, then*
(a.) $InFather[i_k] = i_j$, where $j = Leftmost(SN(k))$.
(b.) If $LCP[t] \leq LCP[t+1]$ or $t = n$ then $LeafFather[i_t] = i_j$, where $j = Leftmost[t]$, otherwise $LeafFather[i_t] = i_{t+1}$.

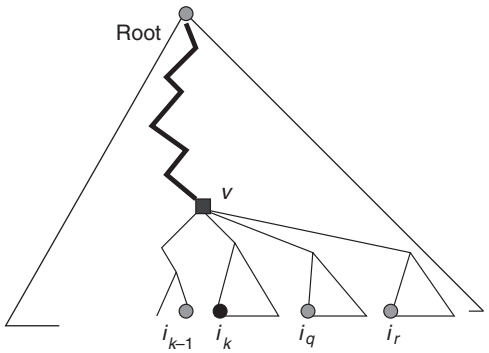

FIGURE 30.13 The node v is identified with i_k, the length of the string $string\_repr(v)$ corresponding to the path from *root* to v equals $LCP[k]$. $Leftmost[r] = Leftmost[q] = Leftmost[k] = k$.

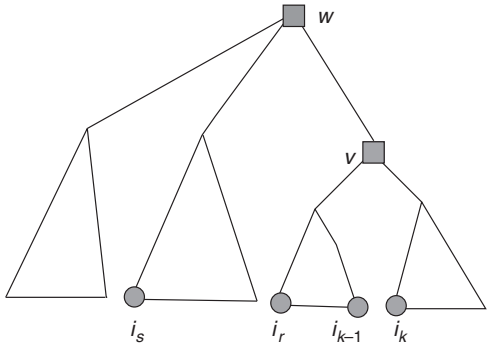

FIGURE 30.14 Computation of the father $w = i_s$ of $v = i_k$. $SN[k] = r$, $Leftmost[r] = s$.

Proof. Define the height of a node v as the length of *string_repr*(v). For an internal node i_k its height equals $LCP[k]$. Assume $w = i_s$ is the father of an internal node v, then all nodes between w and v should have height larger or equal to $height(v)$ and $height(w) < height(v)$, see Figure 30.14. Owing to our representation of internal nodes i_s is the leftmost element to the left of i_k in the suffix array with the same LCP as $SN[k]$. Hence, $s = Leftmost[SN[k]]$. Similar argument works for leaf nodes. ■

Both leaves and internal nodes are identified with integers in the range $[1 \ldots n]$, however, to distinguish between internal node and a leaf with a same number in the implementation we add additional one-bit information.

Algorithm BUILD2(x);

{ The names of internal nods and leaves are integers in $[1 \ldots n]$
{ one-bit information is kept to distinguish internal nodes from leaves}

compute in parallel the tables SN and $Leftmost$;

for each $k \in [1 \ldots n]$
Create leaf node i_k; Create internal node i_k if $Leftmost[k] = k$;

for each node v **do in parallel**
Compute the father w of v according to the formula from Lemma 30.9;
Compute the label of $w \Rightarrow v$ according to formula from Lemma 30.8;

Theorem 30.8 *Assume that the arrays SUF, LCP are given for an input string of size n. The algorithm BUILD2 computes the suffix tree for x on a CREW PRAM in* $O(\log n)$ *time with* $O(n \log n)$ *work and* $O(n)$ *space.*

Proof. The algorithm *BUILD2* is presented informally above. Once the table SN is computed all other computations are computed independently *locally* in $O(\log n)$ time with n processors. ■

The main idea in the construction is to identify the internal nodes of the suffix tree with leaves i_k, such that $v = LCA(i_{k-1}, i_k)$.

The correctness follows from Lemmas 30.7 and 30.9.

30.6 Parallel Construction of Suffix Trees by Refining

The dictionary of basic subwords leads to an efficient alternative construction of suffix trees in $\log^2 n$) time with $O(n)$ processors.

To build the suffix tree of a text, a coarse approximation of it is first built. Afterward the tree is refined step by step, see Figure 30.15. We build a series of a logarithmic number of trees $T_n, T_{n/2}, \ldots, T_1$: each successive tree is an approximation of the suffix tree of the text; the key invariant is:

inv(k): for each internal node v of T_k there are no two distinct outgoing edges for which the labels have the same prefix of the length k; the label of the path from the root to leaf i is $y[i..i+n]$; there is no internal node of outdegree one.

Remark If *inv*(1) holds, then, the tree T_1 is essentially the suffix tree for x. Just a trivial modification may be needed to delete all #'s padded for technical reasons, but one. Note that the parameter k is always a power of two. This gives the logarithmic number of iterations.

The core of the construction is the procedure *REFINE*(k) that transforms T_{2k} into T_k. The procedure maintains the invariant: if *inv*$(2k)$ is satisfied for T_{2k}, then *inv*(k) holds for T_k after running *REFINE*(k) on T_{2k}. The correctness (preservation of invariant) of the construction is based on the trivial fact expressed graphically in Figure 30.15. The procedure *REFINE*(k) consists of two stages:

1. Insertion of new nodes, one per each nonsingleton k-equivalence class
2. Deletion of nodes of outdegree one (*chain nodes*)

We need the following procedure of **local refining**:

procedure *REFINE*(k);
  for each internal node v of T **do** *LocalRefine*(k, v);
  delete all nodes of outdegree one;

The informal description of the construction of the suffix tree is summarized by the algorithm below:

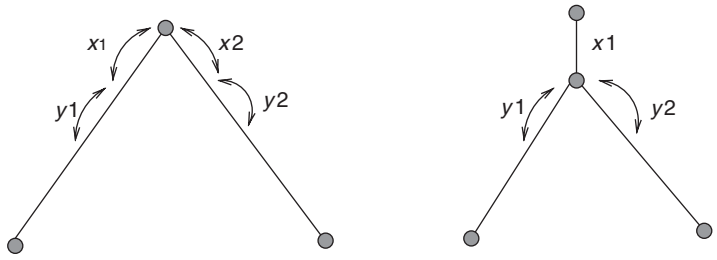

FIGURE 30.15 Insertion of a new node to the temporary suffix tree. We have $|x1| + |y1| = |x2| + |y2| = 2k$ and $|x1| = |x2| = |y1| = |y2|$. We have $|x1| = |x2|$, $y1 \neq |y2|$. After insertion of a new node, if *inv*$(2k)$ is locally satisfied on the left, *inv*(k) holds locally on the right.

```
Algorithm Suffix-Tree-by-Refining;
    let T be the tree of height 1 which leaves are 1, 2,...,n,
    the label of the i-th edge is y[i. .n] encoded as [i, *];
    k := n;
    repeat { T satisfies inv(k) }
        k := k/2;
        REFINE(k);
        {insert in parallel new nodes,
        then remove nodes of outdegree 1;}
    until k = 1;
```

In the first stage the operation *LocalRefine*(k, v) is applied to all internal nodes v of the current tree.

This local operation is graphically presented in Figure 30.15. The k-equivalence classes, labels of edges outgoing a given node, are computed. For each nonsingleton class, we insert a new (internal) node.

The algorithm is informally presented on the example text *abaabbaa#*, see Figures 30.16 and 30.17.

We start with the tree T_8 of all subwords of length 8 starting at positions $1, 2, \ldots, 8$. The tree is almost a suffix tree: the only condition that is violated is that the root has two distinct outgoing edges in which labels have a common nonempty prefix. We attempt to satisfy the condition by successive refinements: the prefixes violating the condition become smaller and smaller, divided by two at each stage, until the final tree is obtained.

Bibliographic Notes

The discussion of general models and techniques related to PRAM's can be found in References 11 and 45. The basic books on string algorithms are References 2, 3, 7, and 18.

The first optimal parallel algorithm for string matching was presented by Galil in Reference 41 for constant size alphabets. Vishkin improved on the notion of the duels of Galil, and described the more powerful concept of (fast) duels that leads to an optimal algorithm independently of the size of the alphabet [60]. The idea of witnesses and duels is used also by Vishkin in Reference 61 in the string matching by sampling. The concept of deterministic sampling is very powerful. It has been used by Galil to design a constant-time optimal parallel searching algorithm (the preprocessing is not included). This result was an improvement upon the $O(\log^* n)$ result of Vishkin, though $O(\log^* n)$ time can also be

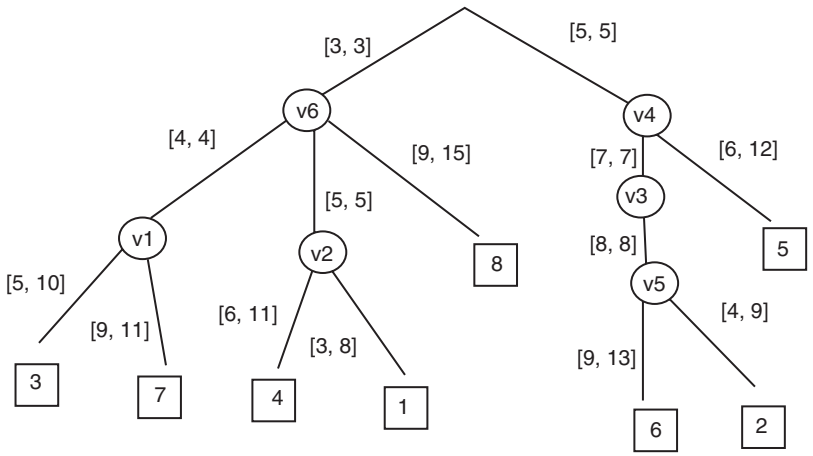

FIGURE 30.16 Tree T_1 after the first stage of *REFINE*(1): insertion of new nodes v_4, v_5, v_6.

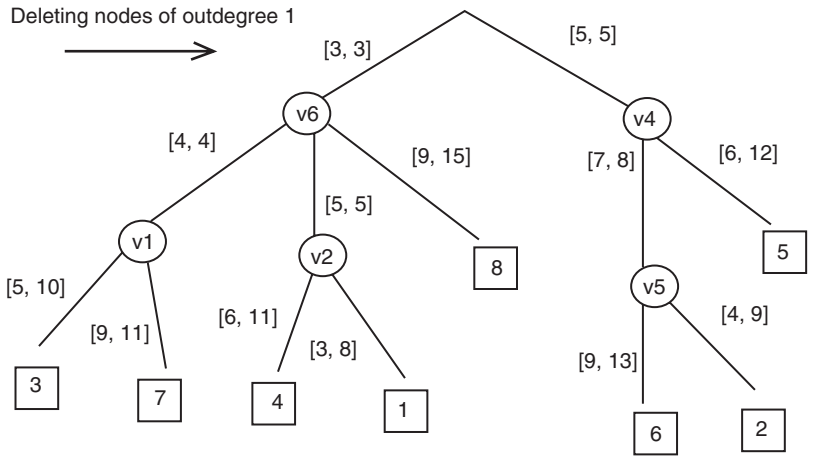

FIGURE 30.17 Tree T_1 after the second stage of *REFINE*(1): deletion of chain nodes.

treated practically as a constant time. There are other parallel-string matching algorithms on different models of computation (CRCW) or randomized ones [30, 31, 40, 42].

Two-dimensional string matching was considered in References 21, 22, 34, and 38.

Approximate string matching has been considered in References 24, 52, 55, and **?**.

AQ: Please provide the Reference Number.

The Karp–Miller–Rosenberg naming technique is from Reference 48. It has been extended in Reference 59. The dictionary of basic subwords and application of Karp–Miller–Rosenberg naming technique in parallel setting is from References 37 and 48.

Suffix arrays were considered in sequential setting in References 4, 5, 12–14, and 16. The parallel sorting algorithms on CREW PRAM working in logarithmic time and $O(n \log n)$ work is from Reference 33.

The fundamental algorithms for suffix trees are in References 1, 8, 17, 19, and 20. The construction of suffix tree by refining was presented in Reference 29. Other parallel algorithms for suffix trees are in References 9, 10, 15, 35, and 46.

The parallel algorithm computing the edit distance is from Reference 24. It was also observed independently in Reference 55. There are other problems for strings considered in parallel settings, see References 23, 25, 36, 42, 54, 56, 57, and 58.

References

[1] Apostolico, A., The myriad virtues of suffix trees, in: Apostolico, A., and Galil, Z., eds., *Combinatorial Algorithms on Words*, NATO Advanced Science Institutes, Series F, Vol. 12, Springer-Verlag, Berlin, 1985.

[2] Charras, C. and Lecroq, T., *Handbook of Exact String Matching Algorithms*, King's College Publications (February 2004).

AQ: Please provide page range.

[3] Crochemore, M., and Rytter, W., *Jewels of Stringology: Text Algorithms*, World Scientific (2003).

[4] Farach, M., *Optimal Suffix Tree Construction with Large Alphabets*, FOCS 1997.

AQ: Please provide place of publication for References 2,3,4,7, 11,12,18.

[5] Farach, M., Ferragina, P., and Muthukrishnan, S., On the sorting complexity of suffix tree construction, *Journal of the ACM*, 47(6): 987–1011, 2000.

[6] Gibbons, A., and Rytter, W., *Efficient Parallel Algorithms*, Cambridge University Press, 1988.

[7] Gusfield, D., *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge Univ. Press, 1997.

[8] Grossi, R., and Italiano, G., Suffix trees and their applications in string algorithms, Techn. Report CS-96-14, Universita di Venezia, 1996.

[9] Hariharan, R., Optimal parallel suffix tree construction. *STOC* 1994: 290–299.

[10] Iliopoulos, C., Landau, G.M., Schieber, B., and Vishkin, U., Parallel construction of a suffix tree with applications, *Algorithmica* 3 (1988): 347–365.

[11] JaJa, J., *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.
[12] Karkkainen, J., and Sanders, P., *Simple Linear Work Suffix Array Construction*, ICALP 2003.
[13] Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K., Linear-time longest-common-prefix computation in suffix arrays and its applications, in *Proc. 12th Combinatorial Pattern Matching*, 181–192.
[14] Ko, P., and Aluru, S., Space efficient linear time construction of suffix arrays, Combinatorial Pattern Matching 2003.
[15] Landau, G.M., Schieber, B., and Vishkin, U., Parallel construction of a suffix tree, in *Automata, Languages and Programming*, Lecture Notes in Computer Science 267, Springer-Verlag, Berlin, 1987, 314–325.
[16] Manber, U. and Myers, E., Suffix arrays: A new method for on-line string searches, in: *Proc. of Ist ACM-SIAM Symposium on Discrete Algorithms*, American Mathematical Society, Providence, R.I., 1990: 319–327.
[17] McCreight, E.M., A space-economical suffix tree construction algorithm, *J. ACM* 23(2) (1976): 262–272.
[18] Smyth, B., *Computing Patterns in Strings*, Pearson Addison Wesley (2003).
[19] Ukkonen, E., Constructing suffix trees on-line in linear time, in IFIP'92, 484–492.
[20] Weiner, P., Linear pattern matching algorithms, in *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, Washington, DC, 1973, 1–11.
[21] Amir, A., Benson, G., and Farach, M., Optimal parallel two-dimensional pattern matching, in *5th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM Press, 1993, 79–85.
[22] Amir, A., and Landau, G.M., Fast parallel and serial multidimensional approximate array matching, *Theoret. Comput. Sci.*, 81 (1991): 97–115.
[23] Apostolico, A., Fast parallel detection of squares in strings, *Algorithmica* 8 (1992): 285–319.
[24] Apostolico, A., Atallah, M.J., Larmore, L.L., and McFaddin, H.S., Efficient parallel algorithms for string editing and related problems, *SIAM J. Comput.*, 19(5) (1990): 968–988.
[25] Apostolico, A., Breslauer, D., and Galil, Z., Optimal parallel algorithms for periods, palindromes and squares, in *3rd Proceedings of SWAT, Lecture Notes in Computer Science 621*, Springer-Verlag, 1992, 296–307.
[26] Berkman, O., Breslauer, D., Galil, Z., Schieber, B., and Vishkin, U., Highly parallelizable problems, in *Proc. 21st ACM Symposium on Theory of Computing, Association for Computing Machinery*, New York, 1989, 309–319.
[27] Apostolico, A., and Crochemore, M., Fast parallel Lyndon factorization and applications, *Math. Syst. Theory* 28(2) (1995): 89–108.
[28] Apostolico, A., and Guerra, C., The longest common subsequence problem revisited, *J. Algorithms* 2 (1987): 315–336.
[29] Apostolico, A., Iliopoulos, C., Landau, G.M., Schieber, B., and Vishkin, U., Parallel construction of a suffix tree with applications, *Algorithmica* 3 (1988): 347–365.
[30] Breslauer, D., and Gall, Z., An optimal $O(\log \log n)$-time parallel string-matching, *SIAM J. Comput.*, 19(6) (1990): 1051–1058.
[31] Chlebus, B.S., Leszek Gasieniec: Optimal pattern matching on meshes. *STACS* 1994: 213–224.
[32] Capocelli, R., ed., *Sequences: Combinatorics, Compression, Security and Transmission*, Springer-Verlag, New York, 1990.
[33] Cole, R., Parallel merge sort, *SIAM J.Comput.*, 17 (1988): 770–785.
[34] Cole, R., Crochemore, M., Galil, Z., Gasieniec, L., Hariharan, R., Muthakrishnan, S., Park, K., and Rytter, W., Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions, in *FOCS'93*, 1993, 248–258.
[35] Crochemore, M., and Rytter, W., Parallel construction of minimal suffix and factor automata, *Inf. Process. Lett.*, 35 (1990): 121–128.
[36] Crochemore, M., and Rytter, W., Efficient parallel algorithms to test square-freeness and factorize strings, *Inf. Process. Lett.*, 38 (1991): 57–60.

[37] Crochemore, M. and Rytter, W., Usefulness of the Karp Miller Rosenberg algorithm in parallel computations on strings and arrays, *Theoret. Comput. Sci.*, 88 (1991): 59–82.

[38] Crochemore, M. and Rytter, W., Note on two-dimensional pattern matching by optimal parallel algorithms, in *Parallel Image Analysis*, Lecture Notes in Computer Science 654, Springer-Verlag, 1992, 100–112.

[39] Crochemore, M., Galil, Z., Gasieniec, L., Park, K., and Rytter , W., Constant-time randomized parallel string matching. *SIAM J. Comput.*, 26(4): (1997): 950–960.

[40] Czumaj, A., Galil, Z., Gasieniec, L., Park, K., and Plandowski, W., Work-time-optimal parallel algorithms for string problems. *STOC* 1995: 713–722.

[41] Galil, Z., Optimal parallel algorithm for string matching, *Information and Control* 67 (1985): 144–157.

[42] Czumaj, A., Gasieniec, L., Piotrw, M., and Rytter, W., Parallel and sequential approximations of shortest superstrings. *SWAT* 1994: 95–106.

[43] Galil, Z., and Giancarlo, R., Parallel string matching with k mismatches, *Theoret Comput. Sci.*, 51 (1987) 341–348.

[44] Gasieniec, L. and Park, K., Work-time optimal parallel prefix matching (Extended Abstract). *ESA* 1994: 471–482.

[45] Gibbons, A. and Rytter, W., *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, U.K., 1988.

[46] Hariharan, R., Optimal parallel suffix tree construction. *J. Comput. Syst. Sci.*, 55(1) (1997): 44–69.

[47] Jàjà, J., *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, MA, 1992.

[48] Karp, R.M., Miller, R.E., and Rosenberg, A.L., Rapid identification of repeated patterns in strings, arrays and trces, in *Proc. 4th ACM Symposium on Theory of Computing, Association for Computing Machinery*, New York, 1972, 125–136.

[49] Kedem, Z.M., Landau, G.M., and Palem, K.V., Optimal parallel suffix-prefix matching algorithm and applications, in *Proc. ACM Symposium on Parallel Algorithms, Association for Computing Machinery*, New York, 1989, 388–398.

[50] Kedem, Z.M. and Palem, K.V., Optimal parallel algorithms for forest and term matching, *Theoret. Comput. Sci.*, 93(2) (1989): 245–264.

[51] Landau, G.M., Schieber, B., and Vishkin, U., Parallel construction of a suffix tree, in *Automata, Languages and Programming*, Lecture Notes in Computer Science 267, Springer-Verlag, Berlin, 1987, 314–325.

[52] Landau, G.M. and Vishkin, U., Introducing efficient parallelism into approximate string matching, in (STOC, 1986): 220–230.

[53] Landau, G.M. and Vishkin, U., Fast parallel and serial approximate string matching, *J. Algorithms*, 10 (1989): 158–169.

[54] Robert, Y. and Tchuente, M., A systolic array for the longest common subsequence problem, *Inf. Process. Lett.*, 21 (1885): 191–198.

[55] Rytter, W., On efficient computations of costs of paths of a grid graph, *Inf. Process. Lett.*, 29 (1988): 71–74.

[56] Rytter, W., On the parallel transformations of regular expressions to non-deterministic finite automata, *Inf. Process. Lett.*, 31 (1989): 103–109.

[57] Rytter, W., and Diks, K., On optimal parallel computations for sequences of brackets, in: [32]: 92–105.

[58] Schieber, B. and Vishkin, U., On finding lowest common ancestors: simplification and parallelization, *SIAM J. Comput.*, 17 (1988): 1253–1262.

[59] Suleyman Cenk Sahinalp and Uzi Vishkin: Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). *FOCS* 1996: 320–328.

[60] Vishkin, U., Optimal parallel pattern matching in strings, *Inf. and Control* 67 (1985): 91–113.

[61] Vishkin, U., Deterministic sampling, a new technique for fast pattern matching, *SIAM J. Comput.*, 20(1) (1991): 22–40.