Basic Research in Computer Science

BRICS RS-95-16 Breslauer & Hariharan: Parallel Construction of Suffix and Factor Automata

Optimal Parallel Construction of Minimal Suffix and Factor Automata

Dany Breslauer
Ramesh Hariharan

BRICS Report Series **RS-95-16**

ISSN 0909-0878 **February 1995**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

Optimal Parallel Construction of Minimal Suffix and Factor Automata

Dany Breslauer* Ramesh Hariharan†

Abstract

This paper gives optimal parallel algorithms for the construction of the smallest deterministic finite automata recognizing all the suffixes and the factors of a string. The algorithms use recently discovered optimal parallel suffix tree construction algorithms together with data structures for the efficient manipulation of trees, exploiting the well known relation between suffix and factor automata and suffix trees.

1 Introduction

Blumer et al. [4] showed that the size of partial deterministic finite automata that recognize the suffixes and the factors (substrings) of a given string is linear in the length of the string and independent of the alphabet size. Blumer et al. [3] and Crochemore [5] gave linear-time on-line algorithms for the construction of the smallest deterministic finite automata recognizing the suffixes and the factors of a string. Crochemore and Rytter [6] gave parallel algorithms for the construction of these automata. Their algorithms take $O(\log n)$ time and use superlinear space on an n-processor CRCW-PRAM. All the algorithms mentioned above exploit in some way or another the close relation between these automata and the suffix tree of the reversed input string.

The time-processor product is an important measure for the efficiency of parallel algorithms. An algorithm with time-processor product (*work*, *operations*) that is equivalent to that of the fastest sequential algorithm for the same problem is said to achieve an *optimal-speedup*, or to be *optimal*. Motivated by the recent discovery of optimal parallel algorithms for the construction of suffix trees, we show in this paper that for strings drawn from a constant sized alphabet, given the suffix tree of the reverse input string, it is possible to construct the minimal suffix and factor automata in $O(\log n)$ time making $O(n)$ operations and using $O(n)$ space in the CRCW-PRAM. For strings drawn from a general ordered alphabet, we show that given the suffix trees of both the input string and its reverse, it is possible to construct the minimal suffix and factor

*BRICS – Basic Research in Computer Science, Centre of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark. Partially supported by ESPRIT Basic Research Action Program of the EC under contract #7141 (ALCOM II).

†Max Planck Institut für Informatik, Saarbrücken, 66123, Germany.

automata in $O(\log n)$ time making $O(n \log |\Sigma|)$ operations and using $O(n)$ space in the CRCW-PRAM. As these bounds are dominated by those of the known suffix tree construction algorithms, the construction of the minimal suffix and factor automata has the same parallel complexity as the suffix tree construction algorithm being used.

A list of the known concurrent-read PRAM suffix tree construction algorithms is given below (for strings over a constant sized alphabet). Observe that although the algorithms given by Crochemore and Rytter [6] for the construction of the minimal suffix and factor automata use a suffix tree construction algorithm, their algorithms do not benefit directly from using any of the recent optimal parallel suffix tree construction algorithms.

Parallel Suffix Tree Construction Algorithms				
Author(s)	Time	Work	Space	Model
Apostolico et al. [1]	$O(\log n)$	$O(n \log n)$	$O(n^{1+\epsilon})$	CRCW
Sahinalp and Vishkin [11]	$O(\log^2 n)$	$O(n)$	$O(n^{1+\epsilon})$	CRCW
Hariharan [8]	$O(\log^4 n)$	$O(n)$	$O(n)$	CREW
Farach and Muthukrishnan[1] [7]	$O(\log n)$	$O(n)$	$O(n)$	CRCW

The paper is organized as follows. Sections 2 and 3 define the suffix tree and the directed acyclic word graph of a string. Sections 4 and 5 give the construction of the minimal suffix and factor automata. Conclusions and open problems are given in Section 6.

2 Suffix trees

Let $w = w_1 \cdots w_n$ be some string from Σ^*, for an arbitrary alphabet Σ. Denote by ϵ the *empty string*, by $\tilde{w} = w_n \cdots w_1$ the string w reversed, by $\mathcal{F}(w)$ the set of all factors (substrings) of w, and by $\mathcal{S}(w)$ the set of all suffixes of w.

The *suffix tree* $\mathcal{T}_w$ of the string w is a rooted tree with edges and nodes that are labeled with substrings of w. The suffix tree satisfies the following properties:

1. Edges leaving (leading away from the root) any given node are labeled with non-empty strings that start with different symbols.
2. Each node is labeled with the string formed by the concatenation of the edge labels on the path from the root to the node.
3. Each internal (non-leaf) node has at least two descendants. (Except the root which might have one descendant in the degenerate case where all symbols of w are the same.)
4. For each factor $v \in \mathcal{F}(w)$, there exists a node labeled $u \in \mathcal{F}(w)$, such that v is a prefix of u.

[1] Las-Vegas type randomized algorithm.

It is a common practice to work with the suffix tree $\mathcal{T}_{w\$}$, where \$ is a special alphabet symbol that does not appear anywhere in w. This guarantees that the suffix tree has exactly $n+1$ leaves that are labeled with all the distinct suffixes of $w\$$. Observe that the edge and the node labels can be represented by indices into the string w, using constant space for each label.

For any node $v = v_1 \cdots v_k$ in $\mathcal{T}_{w\$}$, except the root, define $s(v)$, the *suffix-link* of the node, to be (a pointer to) the node labeled $v_2 \cdots v_k$. McCreight [10], who introduced suffix-links in his sequential suffix tree construction algorithm, shows that $s(v)$ must also be a node in $\mathcal{T}_{w\$}$. Some of the parallel suffix tree construction algorithms use suffix-links as well. We show next that the suffix-links can be efficiently computed given the suffix tree. We will use the following data structure for the *lowest common ancestor* problem:

Lemma 2.1 *(Schieber and Vishkin [12]) Given an h node rooted tree, it is possible to pre-process the tree in $O(\log h)$ time, $O(h)$ operations and $O(h)$ space, in the EREW-PRAM, such that queries about the lowest common ancestor of any pair of nodes can be answered in constant time by a single processor without modifying the pre-processed data structure.*

Lemma 2.2 *Given the suffix tree $\mathcal{T}_{w\$}$, it is possible to compute the suffix-links for all nodes in $\mathcal{T}_{w\$}$ in $O(\log n)$ time making $O(n)$ operations in the CREW-PRAM.*

Proof: Recall that there is one-to-one correspondence between the leaves of $\mathcal{T}_{w\$}$ and the suffixes of $w\$$. This allows to define an array that will give the leaf in $\mathcal{T}_{w\$}$ that corresponds to each suffix in $\mathcal{S}(w\$)$. Hence, the suffix-links of the leaves can be easily computed by setting the suffix-links of the leaf $w_i w_{i+1} \cdots w_n\$$ to point to the leaf $w_{i+1} \cdots w_n\$$ and the suffix-link of the leaf \$ to point to the root which is labeled ϵ.

Next, apply the pre-processing in Lemma 2.1 to the suffix tree $\mathcal{T}_{w\$}$ which has at most $2n+1$ nodes. Then, compute for each internal node v in $\mathcal{T}_{w\$}$, an arbitrary leaf $l(v)$ that is in the sub-tree rooted at v. This can be done in $O(\log n)$ time and $O(n)$ operation in the EREW-PRAM by a pre-order tour of the tree [9].

Now, compute in parallel the suffix-links of each internal node (except the root) as follows. If $v = v_1 \cdots v_k$ is an internal node in $\mathcal{T}_{w\$}$, then it has at least two descendants $y = y_1 \cdots y_h$ and $z = z_1 \cdots z_g$, both have prefix v and $y_{k+1} \neq z_{k+1}$. v is clearly the lowest common ancestor of y and z, and therefore, also of $l(y)$ and $l(z)$. $s(v)$ is the lowest common ancestor of $s(y)$ and $s(z)$. But $s(l(y))$ and $s(l(z))$ are nodes in the sub-trees rooted at $s(y)$ and $s(z)$, respectively, and therefore, $s(v)$ is also the lowest common ancestor of $s(l(y))$ and $s(l(z))$. Recall that $s(l(y))$ and $s(l(z))$ were already computed since $l(y)$ and $l(z)$ are leaves. Hence, the lowest common ancestor of $s(l(y))$ and $s(l(z))$ can be found in constant time using a single processor, by Lemma 2.1. Since many lowest common ancestor queries are processed in parallel, we need the CREW-PRAM model. □

Define the *extended suffix tree* $\hat{\mathcal{T}}_w$ to be the same as the suffix tree $\mathcal{T}_w$ with the exception that there is a node labeled with every suffix of w. This allows nodes that are labeled with suffixes of w to have only one descendant. It is not difficult to see that $\hat{\mathcal{T}}_w$ is an intermediate between $\mathcal{T}_w$ and $\mathcal{T}_{w\$}$. It can be obtained from $\mathcal{T}_w$ by breaking up edges and introducing nodes that correspond to suffixes of w, or it can be obtained from $\mathcal{T}_{w\$}$ by deleting all the leaves that the edges leading to them are labeled with \$.

Lemma 2.3 *It is possible to construct the extended suffix tree $\hat{\mathcal{T}}_w$ and its suffix-links, given the suffix tree $\mathcal{T}_{w\$}$ and its suffix-links, in constant time and $O(n)$ operations in the CREW-PRAM.*

Proof: To identify which leaves in $\mathcal{T}_{w\$}$ have to be deleted to obtain $\hat{\mathcal{T}}_w$, it suffices to assign a single processor to each leaf to examine if the edge leading to the leaf is labeled \$. If v is a node in $\hat{\mathcal{T}}_w$, then clearly $s(v)$ is also a node. To identify which suffix-link pointers have to be changed from $\mathcal{T}_{w\$}$ to $\hat{\mathcal{T}}_w$, observe that if the suffix $w_i \cdots w_n$ is an internal node, then its suffix-link pointer does not change, since $w_{i+1} \cdots w_n$ is also an internal node. This characterizes precisely those leaves in $\mathcal{T}_{w\$}$ that are not leaves in $\hat{\mathcal{T}}_w$. Let $w_h \cdots w_n$ be the longest suffix of w that occurs at least twice in w (letting $h = n+1$ if there is no such non-empty suffix and taking $w_{n+1} = \$$). Then, the leaf $w_g \cdots w_n\$$ is deleted if and only if $h \leq g \leq n+1$. The only suffix-link pointer in $\mathcal{T}_{w\$}$ that has to be modified is the suffix-link of the leaf $w_{h-1} \cdots w_n$ which is set to point to the internal node $w_h \cdots w_n$. This leaf can be identified as the only leaf in $\mathcal{T}_{w\$}$ which is not deleted and whose suffix-link is deleted. □

Finally, we will need the following processing in order to facilitate the computation in Section 3.

Lemma 2.4 *$\mathcal{T}_{w\$}$ and $\mathcal{T}_{\tilde{w}\$}$ can be pre-processed in $O(\log n)$ time and $O(n)$ work on the EREW-PRAM so that given an internal node $\tilde{u}$ in $\mathcal{T}_{\tilde{w}\$}$, the node u in $\mathcal{T}_{w\$}$, if it exists, can be found in constant time by a single processor.*

Proof: The pre-processing for $\mathcal{T}_{\tilde{w}\$}$ consists of the following steps. First, an array $\mathcal{L}$ containing the leaves of $\mathcal{T}_{\tilde{w}\$}$ in order is computed. Second, for each internal node in $\mathcal{T}_{\tilde{w}\$}$, the offsets of the first and the last leaves in $\mathcal{L}$ which lie in the subtree rooted at that node are computed. Third, for a leaf l of $\mathcal{T}_{\tilde{w}\$}$, let $prev(l)$ be the character in $\tilde{w}\$$ which immediately precedes the suffix of $\tilde{w}\$$ corresponding to l; then, for each leaf l, the nearest leaf $next(l)$ to its right in $\mathcal{L}$ such that $prev(next(l)) \neq prev(l)$ is found. Each of the above three steps can easily be accomplished in $O(\log n)$ time and $O(n)$ work on the EREW-PRAM.

The pre-processing for $\mathcal{T}_{w\$}$ consists only of the lowest common ancestor pre-processing of Lemma 2.1.

Next, suppose we are given internal node $\tilde{u}$ in $\mathcal{T}_{\tilde{w}\$}$. Note that u is a node in $\mathcal{T}_{w\$}$ if and only if there exist two leaves l, l' in the subtree of $\mathcal{T}_{\tilde{w}\$}$ rooted at $\tilde{u}$ such that $prev(l) \neq prev(l')$. This happens, in turn, if and only if $next(l'')$ is in the above subtree, where l'' is the leftmost leaf in this subtree. Clearly, this can be checked in constant time and work using the above pre-processing. If $next(l'')$

is indeed in the above subtree, then l'' and $next(l'')$ give two occurrences of u in w such that the characters following these two occurrences of u are distinct. Finding the lowest common ancestor in $\mathcal{T}_{w\$}$ of the leaves corresponding to the two suffixes of $w\$$ beginning at these two occurrences of u completes the task in constant time and work. □

3 The directed acyclic word graph

The discussion in this section follows Blumer et al. [3]. Define the *end-set* of a string u in w to be the set of all ending positions of occurrences of u in w. Formally, $end\text{-}set_w(u) = \left\{h \mid u = w_{h-|u|+1} \cdots w_h\right\}$ and $end\text{-}set_w(\epsilon) = \{1, \ldots, |w|\}$. Two strings u and v are said to be *end-equivalent* in w, denoted $u \equiv_w v$, if $end\text{-}set_w(u) = end\text{-}set_w(v)$. Denote by $[u]_w$ the equivalence class of u with respect to $\equiv_w$. The class containing all strings that are not in $\mathcal{F}(w)$ is called the *degenerate class*. We choose the longest member in each equivalence class to be the *canonical representative* of the equivalence class. See the paper by Blumer et. al. [3] for more detail.

Definition 3.1 *The* Directed Acyclic Word Graph *(DAWG) for the string w is the directed graph whose set of nodes is the non-degenerate equivalence classes $\{[u]_w \mid u \in \mathcal{F}(w)\}$ and set of edges, which are labeled with alphabet symbols, is $\{[u]_w \rightarrow^a [ua]_w \mid u, ua \in \mathcal{F}(w)\}$. (One can easily verify that this definition does not depend on the representatives of the equivalence classes.)*

The DAWG of w can be viewed as a partial deterministic finite automaton with initial state $[\epsilon]_w$ and all states being accepting states. This automaton recognizes exactly the strings in $\mathcal{F}(w)$ [3]. In the rest of this section we describe a new efficient parallel construction of the DAWG of a string.

The relation between the DAWG of w and the suffix tree of $\tilde{w}$ has been established in [3, 4, 5], where it is shown that the canonical representatives of non-degenerate equivalence classes, which correspond to the nodes in the DAWG, are exactly the *reversed labels* of the nodes in $\hat{\mathcal{T}}_{\tilde{w}}$, and that the number of edges in the DAWG is at most by n larger than the number of nodes, independent of the alphabet.

Given the extended suffix tree $\hat{\mathcal{T}}_{\tilde{w}}$, we can copy its nodes to be the nodes of the DAWG of w, which leaves the problem of finding the edges of the DAWG. We will use the following data structure for the *level-ancestor* problem:

Lemma 3.2 *(Berkman and Vishkin [2]) Given an h node rooted tree, it is possible to pre-process the tree in $O(\log h)$ time, $O(h)$ operations and $O(h)$ space, in the CRCW-PRAM, such that level-ancestor queries that find the l-th node on the path from the node v to the root can be answered in constant time by a single processor without modifying the pre-processed data structure.*

Theorem 3.3 *For strings w drawn from a constant sized alphabet, there exists an $O(\log n)$ time, $O(n)$ operations CRCW-PRAM algorithm that constructs the DAWG for w given the extended suffix tree $\hat{\mathcal{T}}_{\tilde{w}}$ and its suffix-links. The same can*

be accomplished for strings drawn from a general alphabet, making $O(n \log |\Sigma|)$ operations, provided that $\mathcal{T}_{w\$}$ is also given.

Proof: As mentioned above, the nodes of the DAWG are exactly the nodes of $\hat{\mathcal{T}}_{\tilde{w}}$ and it remains to compute the edges of the DAWG. Recall that if some canonical representative u is a node in the DAWG, then $\tilde{u}$ is a node in $\hat{\mathcal{T}}_{\tilde{w}}$.

Let u be a node in the DAWG and $a \in \Sigma$, such that $ua \in \mathcal{F}(w)$. Then, there is an edge $[u]_w \rightarrow^a [ua]_w$ in the DAWG. Let v be the canonical representative of $[ua]_w$. If $v = ua$, then $\tilde{v} = a\tilde{u}$ is a node in $\hat{\mathcal{T}}_{\tilde{w}}$ and its suffix-link $s(a\tilde{u}) = \tilde{u}$ is exactly the DAWG edge, reversed. However, in general, this is not necessarily the case.

Let $p(x)$ be the immediate ancestor of x in $\hat{\mathcal{T}}_{\tilde{w}}$ and let $d(x)$ denote the depth of x in $\hat{\mathcal{T}}_{\tilde{w}}$. The depth of all nodes can be computed in $O(\log n)$ time and $O(n)$ operations in the EREW-PRAM [9]. Let $\tilde{z_1} = s(\tilde{v}), \tilde{z_2}, \ldots, \tilde{z_h}$, for $h = d(s(\tilde{v})) - d(s(p(\tilde{v})))$, be the suffix-link of $\tilde{v}$ and its ancestors, up to and excluding the suffix-link of $p(\tilde{v})$. Then, $z_1, \ldots, z_h$ are nodes in the DAWG, all with edges $[z_i]_w \rightarrow^a [v]_w$. This characterizes all the edges in the DAWG.

The computation of the edges proceeds as follows. A processor assigned to each node $\tilde{v}$ in $\hat{\mathcal{T}}_{\tilde{w}}$ computes the number of edges that will be coming into the corresponding node v in the DAWG. There are exactly $d(s(\tilde{v})) - d(s(p(\tilde{v})))$ such edges. By summing the number of edges, processors can be assigned to compute each edge. This takes $O(\log n)$ time and $O(n)$ operations in the EREW-PRAM.

The processors that are assigned to a node to compute its incoming edges still have to find the nodes on the other side of these edges. After applying the pre-processing in Lemma 3.2, these nodes can be found by level ancestor queries, since they all are ancestors of $s(\tilde{v})$.

Finally, we need to organize the outgoing edges for each node in DAWG. This is easily done in constant time and $O(n)$ work when the alphabet size is constant, as every node has a constant sized array representing its outgoing edges which are updated directly within this array. Consider the case of a larger alphabet. In this case, we use the suffix tree of the input string w to organize the edges leaving a given node. We rely on the fact that a node z in the DAWG has two or more outgoing edges if and only if z is a node in $\mathcal{T}_{w\$}$.

Recall from above that associated with node $\tilde{v}$ in $\hat{\mathcal{T}}_{\tilde{w}}$ is a group of processors, one for each of the nodes $\tilde{z}_1, \tilde{z}_2, \ldots, \tilde{z}_h$. Let $\mathcal{P}_i$ be the processor associated with $\tilde{z}_i$. $\mathcal{P}_i$ executes the following sequence of operations.

First, $\mathcal{P}_i$ checks if z_i is a node in $\mathcal{T}_{w\$}$ using Lemma 2.4 (for this, note that each node in $\hat{\mathcal{T}}_{\tilde{w}}$ corresponds to a unique node in $\mathcal{T}_{\tilde{w}\$}$). This takes constant time and work on the CREW-PRAM model. There are two cases next. If z_i is not a node in $\mathcal{T}_{w\$}$ then it can be easily seen that node z_i in DAWG has only one outgoing edge, namely to v; in this case, $\mathcal{P}_i$ simply writes this edge into z_i. Suppose that z_i is a node in $\mathcal{T}_{w\$}$.

We assume that associated with every node y in $\mathcal{T}_{w\$}$ is a sorted array $\mathcal{A}_y$, whose locations correspond to the various characters which are the first characters of the substrings labeling edges leading from y to its children. Note that the sum of the sizes of the arrays $\mathcal{A}_y$ over all nodes y of $\mathcal{T}_{w\$}$ is just the size of $\mathcal{T}_{w\$}$, i.e., $O(n)$. Let a be the first character of $\tilde{v}$. $\mathcal{P}_i$ finds the location in array

$\mathcal{A}_{z_i}$ corresponding to character a and adds a pointer to the node v of DAWG at this location; this takes $O(\log|\Sigma|)$ time and work.

At the end, the set of edges leaving z_i in DAWG is exactly the set of pointers in $\mathcal{A}_{z_i}$. The above procedure takes $O(\log n)$ time and $O(n\log|\Sigma|)$ work on the whole. □

4 The suffix automaton

The minimal suffix automata, henceforth denoted MSA, is characterized next:

Lemma 4.1 *(Crochemore [5]) The MSA is the DAWG except that the accepting states are those equivalence classes that include suffixes of w.*

Theorem 4.2 *The MSA, recognizing the strings in $\mathcal{S}(w)$, can be constructed in $O(\log n)$ time, making $O(n)$ operations and using $O(n)$ space in the CRCW-PRAM, given the extended suffix tree $\hat{\mathcal{T}}_{\tilde{w}}$.*

Proof: By Theorem 3.3, the DAWG can be constructed within these bounds. So, by Lemma 4.1, it remains to identify the accepting states. Observe that u is a suffix of w if and only if $\tilde{u}$ is a prefix of $\tilde{w}$. Then, a node $\tilde{u}$ in $\hat{\mathcal{T}}_{\tilde{w}}$ is a prefix of $\tilde{w}$ if and only if it is an ancestor of the node $\tilde{w}$ in $\hat{\mathcal{T}}_{\tilde{w}}$. These nodes can be identified by pre-order and post-order tours of $\hat{\mathcal{T}}_{\tilde{w}}$ in $O(\log n)$ time and $O(n)$ operations in the EREW-PRAM [9]. □

5 The factor automaton

The next lemma follows from [3, 6] and states the relationship between the DAWG and the minimal factor automaton, henceforth denoted by MFA. Here, let z be the longest suffix of w, if any, which occurs more than once in w. Note that z may not be defined. Let a be the character preceding suffix z in w, i.e., az is also a suffix of w.

Lemma 5.1 *States u and v in the DAWG must be represented by the same state in MFA if and only if:*

1. *z is defined.*
2. *u is a prefix of z.*
3. *Node $\tilde{u}$ in $\hat{\mathcal{T}}_{\tilde{w}}$ has exactly two children, one of which, namely $\tilde{u}\tilde{y}$, is a leaf, where $w = yz$ and the first symbol of y is a.*
4. *Node $\tilde{v}$ in $\hat{\mathcal{T}}_{\tilde{w}}$ is the child of $\tilde{u}$ such that $\tilde{u}a$ is not a prefix of $\tilde{v}$.*

It follows from Lemma 5.1 that the MFA can be obtained by identifying all pairs of states u and v in the DAWG which are represented by the same state in MFA. An important fact to note is that all these pairs are disjoint; this can be easily seen from Lemma 5.1.

The algorithm for obtaining the MFA from the DAWG is as follows.

Step 1. Determine z and find two occurrences of z in w. $\tilde{z}$ is simply the parent of $\tilde{w}$ in $\hat{\mathcal{T}}_{\tilde{w}}$, so z is easily determined. If z is not defined (i.e., $\tilde{w}$ is a child of the root), then nothing further needs to be done. Otherwise, one occurrence of $\tilde{z}$ is simply as a prefix of $\tilde{w}$. If $\tilde{z}$ is also a suffix of $\tilde{w}$ then another occurrence of $\tilde{z}$ is found, otherwise the node $\tilde{z}$ has a child $\tilde{y}$ in addition to $\tilde{w}$; in this case the leaf $l(\tilde{y})$ gives another occurrence of $\tilde{z}$. Clearly, the above step takes constant time and work on the CREW-PRAM.

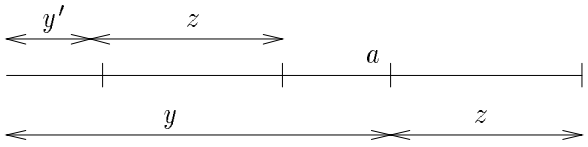

Figure 1: Two Occurrences of z.

In order to denote the above two occurrences of z, we let y, y' be such that $w = yz$ and $y'z$ is a prefix of w. Note that y ends with an a while y' cannot end with an a and could possibly be the empty string.

Step 2. Determine the u, v pairs as follows. All prefixes u of z are processed in parallel. Consider one such prefix u. The node $\tilde{u}$ is found by computing the lowest common ancestor of $\tilde{u}\tilde{y}'$ and $\tilde{u}\tilde{y}$ in $\hat{\mathcal{T}}_{\tilde{w}}$. If the leaf $\tilde{u}\tilde{y}$ is a child of $\tilde{u}$ and $\tilde{u}$ has exactly two children, then $\tilde{v}$ is the other child of $\tilde{u}$ and a u, v pair is found. This step takes constant time and $O(n)$ operations on the CREW-PRAM.

Step 3. Merge the u, v pairs found above. These two states will be represented by one state in the MFA. Merging them involves merging the lists of incident edges in the DAWG. Since all the pairs are disjoint, this takes $O(\log n)$ time and $O(n)$ work (recall the DAWG has only $O(n)$ edges) on the CREW-PRAM.

Theorem 5.2 *The MFA, recognizing the strings in $\mathcal{F}(w)$, can be constructed in $O(\log n)$ time, making $O(n)$ operations and using $O(n)$ space in the CREW-PRAM, given the extended suffix tree $\hat{\mathcal{T}}_{\tilde{w}}$. The same can be accomplished for strings drawn from a general alphabet, making $O(n \log |\Sigma|)$ operations, provided that $\mathcal{T}_{w\$}$ is also given.*

6 Conclusion

In this paper we have shown that the minimal suffix and factor automata of a string can be constructed optimally in parallel. It is not known, however, if one of the important features of these automata in sequential computation, namely, the ability to identify substrings and find the information associated with them, in time that is proportional to the length of the substrings, can be done as efficiently in parallel.

Blumer et al. [4] and Crochemore [5] show that building on the factor automata, one can build the *complete inverted file* and the *factor transducer* of a string on-line in linear time. It is not difficult to verify that using standard parallel algorithmic techniques, the same information can be computed optimally in $O(\log n)$ time by the CREW-PRAM.

Finally, it would be interesting to find an optimal implementation of the algorithms presented in this paper in the CREW-PRAM. Notice that we used the stronger CRCW-PRAM model only in the primitive for computing the level ancestors (Lemma 3.2).

References

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

[2] O. Berkman and U. Vishkin. Finding Level-Ancestors in Trees. *J. Comput. System Sci.*, 48:214–230, 1994.

[3] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The Smallest Automaton Recognizing the Subwords of a Text. *Theoret. Comput. Sci.*, 40:31–55, 1985.

[4] A. Blumer, J. Blumer, D. Haussler, R. McConnel, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. Assoc. Comput. Mach.*, 34(3):578–595, 1987.

[5] M. Crochemore. Transducers and repetitions. *Theoret. Comput. Sci.*, 12:63–86, 1986.

[6] M. Crochemore and W. Rytter. Parallel Construction of Minimal Suffix and Factor Automata. *Inform. Process. Lett.*, 35:121–128, 1990.

[7] M. Farach and S. Muthukrishnan. Personal communication, 1994.

[8] R. Hariharan. Optimal Parallel Suffix Tree Construction. In *Proc. 26th ACM Symp. on Theory of Computing*, pages 290–299, 1994.

[9] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA., U.S.A., 1992.

[10] E.M. McCreight. A space economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.*, 23:262–272, 1976.

[11] S.C. Sahinalp and U. Vishkin. Symmetry Breaking for Suffix Tree Construction. In *Proc. 26th ACM Symp. on Theory of Computing*, pages 300–309, 1994.

[12] B. Schieber and U. Vishkin. On finding Lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

Recent Publications in the BRICS Report Series

RS-95-16 **Dany Breslauer and Ramesh Hariharan.** ***Optimal Parallel Construction of Minimal Suffix and Factor Automata*****. February 1995. 9 pp.**

RS-95-15 **Devdatt P. Dubhashi, Grammati E. Pantziou, Paul G. Spirakis, and Christos D. Zaroliagis.** ***The Fourth Moment in Luby's Distribution*****. February 1995. 10 pp.**

RS-95-14 **Devdatt P. Dubhashi.** ***Inclusion–Exclusion***(3) ***Implies Inclusion–Exclusion***(n)**. February 1995. 6 pp.**

RS-95-13 **Torben Braüner.** ***The Girard Translation Extended with Recursion*****. 1995. Full version of paper to appear in Proceedings of CSL '94, LNCS.**

RS-95-12 **Gerth Stølting Brodal.** ***Fast Meldable Priority Queues*****. February 1995. 12 pp.**

RS-95-11 **Alberto Apostolico and Dany Breslauer.** ***An Optimal*** $\boldsymbol{O}(\log\log n)$ ***Time Parallel Algorithm for Detecting all Squares in a String*****. February 1995. 18 pp. To appear in SIAM Journal on Computing.**

RS-95-10 **Dany Breslauer and Devdatt P. Dubhashi.** ***Transforming Comparison Model Lower Bounds to the Parallel-Random-Access-Machine*****. February 1995. 11 pp.**

RS-95-9 **Lars R. Knudsen.** ***Partial and Higher Order Differentials and Applications to the DES*****. February 1995. 24 pp.**

RS-95-8 **Ole I. Hougaard, Michael I. Schwartzbach, and Hosein Askari.** ***Type Inference of Turbo Pascal*****. February 1995. 19 pp.**

RS-95-7 **David A. Basin and Nils Klarlund.** ***Hardware Verification using Monadic Second-Order Logic*****. January 1995. 13 pp.**

RS-95-6 **Igor Walukiewicz.** ***A Complete Deductive System for the*** μ***-Calculus*****. January 1995. 39 pp.**

RS-95-5 **Luca Aceto and Anna Ingólfsdóttir.** ***A Complete Equational Axiomatization for Prefix Iteration with Silent Steps*****. January 1995. 27 pp.**