HAL
open science

Fast parallel Lyndon factorization and applications

Alberto Apostolico, Maxime Crochemore

To cite this version:

Alberto Apostolico, Maxime Crochemore. Fast parallel Lyndon factorization and applications. Mathematical System Theory, 1995, 28 (2), pp.89-108. hal-00619181

HAL Id: hal-00619181

https://hal.science/hal-00619181

Submitted on 27 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Purdue University
Purdue e-Pubs

Computer Science Technical Reports | Department of Computer Science

1989

Fast Parallel Lyndon Factorization With Applications

Alberto Aposiolico

Maxime Crochemore

Report Number:
89-931

Aposiolico, Alberto and Crochemore, Maxime, "Fast Parallel Lyndon Factorization With Applications" (1989). *Computer Science Technical Reports.* Paper 792.
http://docs.lib.purdue.edu/cstech/792

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

FAST PARALLEL LYNDON FACTORIZATION, WITH APPLICATIONS

Alberto Apostolico
Maxime Crochemore

CSD-TR-931
November 1989
(Revised April 1990)

FAST PARALLEL LYNDON FACTORIZATION WITH APPLICATIONS

Alberto Apostolico[1]
Department of Computer Science, Purdue University, West Lafayette, IN 47907
and
Dipartimento di Matematica Pura e Applicata, Università de L'Aquila, L'Aquila, Italy

Maxime Crochemore[2]
L.I.T.P., University of Paris VII, 4-Place Jussieu, 75252 Paris, France

Purdue University CS TR 931
(November, 1989)
(Revised April 1990)

Abstract: It is shown that the Lyndon decomposition of a word of n symbols can be computed by an n-processors CRCW PRAM in $O(\log n)$ time. Extensions of the basic algorithm convey, within the same time and processors bounds, efficient parallel solutions to problems such as finding the lexicographically minimum or maximum suffix for all prefixes of the input string, and finding the lexicographically least rotation of all prefixes of the input.

Key words: parallel computation, combinatorics on words, string matching, Lyndon words.

AMS subject classification: 68C25

[1] This author's research was supported in part by the French and Italian Ministries of Education, by the British Research Council Grant SERC-E76797, by NSF Grant CCR-89-00305, by NIH Library of Medicine Grant R01 LM05118, by AFOSR Grant 90-0107, and by NATO Grant CRG 900293.

[2] This author's research was supported in part by PRC 'Mathématiques et Informatique' and by NATO Grant CRG 900293.

1. INTRODUCTION

Within the vast domain of sorting, a special role is played by problems defined in terms of *lexicographic* orders. Among problems in this class, we find that of sorting a set of strings over some ordered alphabet, finding the lexicographically least circular shift of a string, finding the lexicographically smallest or largest suffix for a string, etc. In the realm of serial computation, the last three problems are solved efficiently by resort to a special factorization of the free monoid [13] introduced in [8] and known as *Lyndon factorization* (or *decomposition*). According to this factorization, any word can be decomposed uniquely into a sequence of lexicographically non increasing factors, with the additional property that each such factor is lexicographically least among its own circular shifts. Optimal, linear-time algorithms for the Lyndon factorization of a word were given in [10], along with the implied linear-time solutions for the related problems of finding lexicographically least circular shifts, computing minimum suffixes, etc. Recently, further properties of the Lyndon factorization were used to compute in optimal $O(n^2)$ time the least rotations of all substrings of a string of n symbols [3].

A handful of algorithmic problems on strings [4] have been attacked to date also in the framework of parallel computation (see, e.g., [2], [5], [6], [9]). In this framework, one is usually interested in approaching *optimum speed-up* for a problem, in the sense that the product of the time taken by a parallel solution and the number of processors used should be as close as possible to the asymptotic complexity of the best serial algorithm available for that problem. Typically, the available efficient sequential algorithms do not lend themselves to efficient parallelizations, so that fast parallel algorithms are to be developed mostly from scratch. In particular, none of the parallel algorithms on words produced resembles any sequential predecessor. The algorithm presented in this paper is no exception to this rule. In this paper, we show that the Lyndon decomposition of a word of n symbols can be computed by an n-processors CRCW PRAM in $O(\log n)$ time and linear space. The best previous parallel solution to this problem uses a CRCW PRAM with n processors and takes $O(\log^2 n)$ time with linear space, or $O(\log n)$ time with quadratic space [9]. Although the *time* $\times$ *processors* bound of this paper does not achieve optimum speed-up, it is very close to the $\Omega(\log n/\log\log n)$ lower bound for computing such elementary functions as the parity of n bits on a CRCW PRAM using a polynomial number of processors [7].

This paper is organized as follows. In the next section, we recall some basic known facts of combinatorics on words and lexicographic orderings. In Section 3, we analyze the robustness of the Lyndon decomposition of a word x under extension operations that change x into a new word $x' = xw$, where w is an arbitrary word. In Section 4, we study more in detail the relation between the Lyndon factorizations of two given words x and x' and the Lyndon decomposition of $w = xx'$. Some of the properties we derive are of

independent interest. Section 5 contains the description of our parallel algorithm for the Lyndon decomposition of a word, based on the results of the previous sections. In the final section, we describe some applications and extensions of the main algorithm that lead to solve, in overall $O(\log n)$ time, some problems defined on the set of all prefixes of the input string. Specifically, we consider the problem of finding, for every prefix of the input, the lexicographically smallest or largest suffix, and the lexicographically least among all rotations of that prefix. The last application is based on some properties of Lyndon decompositions recently introduced in [3].

2. PRELIMINARIES

Let Σ be an alphabet totally ordered according to the relation $<$, Σ^+ the free semigroup generated by Σ, and $\Sigma^* = \Sigma^+ \cup \{\lambda\}$, where λ is the empty word. The total order $<$ is extended in its corresponding *lexicographic order* on Σ^+, as follows: For any pair of words $x, y \in \Sigma^+$, $x < y$ iff either $y \in x\Sigma^+$ or

$$x = ras,\ y = rbt, \text{ with } a < b; a, b \in \Sigma; r, s, t \in \Sigma^*.$$

In the following, we write $u \ll v$ or $v \gg u$ to denote that $u < v$ but v is not in $u\Sigma^*$.

Fact 1 . Let $u \ll v$. Then, for any w, z in Σ^*, we have $uw \ll vz$.

A word $x \in \Sigma^*$ is a *Lyndon word* iff x is strictly smaller that any of its proper suffixes. For example, $a, b, aaab, abbb, aabab$ and $aababaabb$ are Lyndon words on the alphabet $\Sigma = a, b$, but $abab$ and $abaab$ are not. By the definition of lexicographic order, one gets then immediately that if x is a Lyndon word, then no nonempty suffix of x can be also a prefix of x. A word with this property is *border-free* . A word x is *primitive* if setting $x = w^k$ implies $k = 1$. An immediate consequence of the preceding statement is then that any Lyndon word is a primitive word. A word x is *strongly primitive* or *squarefree* if every substring of x is a primitive word. For example, *cabca* and *cababd* are primitive words, but *cabca* is also strongly primitive, while *cababd* is not, due to the *square abab*.

Fact 2. Let l be a Lyndon word, $v \in \Sigma^+$ a suffix of l and u a prefix of v. Then $u < l$ implies that u is a prefix of l. In other words, $u \ll l$ is impossible.

The following central theorem holds [8].

Theorem 1. Any word $x \in \Sigma^+$ may be written in a unique way as a nonincreasing product of Lyndon words:

$$x = l_1 l_2 ... l_k,\ l_1 \geq l_2 \geq ... \geq l_k.$$

Moreover, l_k is the (lexicographically) smallest suffix of x.

The sequence $(l_1, l_2, ..., l_k)$ of Lyndon words such that $x = l_1 l_2 ... l_k$ and $l_1 \geq l_2 \geq ... \geq l_k$ is called the *Lyndon decomposition* or *Lyndon factorization* of x. In the following, we refer to it simply as the *decomposition* or *factorization* of x. The following claim is an obvious consequence of the fact that l_1 is the longest Lyndon word that is also a prefix of x.

Fact 3. Let l be a Lyndon word. For any $w \in \Sigma^*$, the first factor l_1 in the decomposition of lw obeys the condition $|l_1| \geq |l|$.

The following notions will be needed in the sequel. If $x = vwy$, then the length $|v|$ of v is the *position* of w in x. Let $l_1 l_2 ... l_k$ be the Lyndon decomposition of a string x. For any t $(t = 1, 2, ..., k-1)$, $tail(l_t)$ is the suffix of x having the same position in x as l_{t+1}, i.e., $tail(l_t) = l_{t+1} l_{t+2} ... l_k$. We also set $tail(l_k) = \lambda$, and, with the convention that $l_0 = \lambda$, $tail(l_0) = x$. For any $t(t = 1, 2, ..., k)$, $rest(l_t)$ is the suffix of x at position $i + |l_t|$, where i is the position of the last factor identical to l_t in the decomposition of x. For example, for $x = bbababa$ we have $l_1 = l_2 = b$, $l_3 = l_4 = ab$ and $l_5 = a$. We also have $tail(l_1) = bababa$ and $rest(l_1) = tail(l_2) = ababa$. Finally, $tail(l_3) = aba$, and $tail(l_4) = rest(l_4) = rest(l_3) = a$.

3. FACTOR STABILITY UNDER RIGHT EXTENSIONS

In this section, we study the robustness of the factors in the decomposition of a word x with respect to arbitrary extensions of x into a new word xw. It is easily seen that the factorization of some such extensions are themselves easy extensions of the factorization of x, while others depart quite substantially from the factorization of x. For example, let $x = abcababcababcab$. Then, we have $l_1 = abc$, $l_2 = l_3 = ababc$ and $l_4 = ab$. Appending to x a string w consisting of the single symbol b leaves l_1, l_2 and l_3 unaltered, and only requires extending l_4 into the new factor abb. If, on the other hand, we had chosen $w = c$, then the decomposition of xc would have $l_1 = abc$ and $l_2 = ababcababcabc$, which is dramatically different from the decomposition of x.

We say that a factor l in the decomposition of a string x is *right-stable* if l is a factor of the decomposition of xw for any $w \in \Sigma^*$. Let now $l_1, l_2, ..., l_k$ be the decomposition of x. Clearly, l_k is never right stable unless l_k coincides with the maximum symbol c in Σ. However, if $l_k = c$ and $k > 1$, then it must be $l_1 = l_2 = ... = l_k$, and all factors are right-stable. Observe also that, for $|l_i| > 1$ the first symbol of l_i cannot be c. In the nontrivial case that l_k is not c and that $k > 1$, the following theorem characterizes the right-stable factors in the decomposition of x.

Theorem 2. Let $l_1, l_2, ..., l_k$ be the decomposition of x, and assume that $k > 1$ and $l_k \neq c$. For any t in $[1, k-1]$, l_t is right-stable if and only if $rest(l_t)$ is not a prefix of l_t.

Proof. Assume that l_t is right-stable but $rest(l_t)$ is a prefix of l_t. We show that, in this case, l_t is not a factor in the decomposition of xc. By the definition of $rest(l_t)$, we have that $w = tail(l_{t-1}) = (l_t)^f rest(l_t)$ for some integer $f \geq 1$. We show that wc is a Lyndon word. Thus, even if a factor of the decomposition of xc started with l_t, l_t would only be a proper prefix of such a factor. Choose i such that $|rest(l_t)c^i| = |l_t| + 1$. Let z be an arbitrary suffix of l_t, and u be the suffix of of $rest(l_t)c^{i-1}$ such that $|u| = |z|$. Clearly, $u \geq z$, whence $uc > z$. But $z \gg l_t$, so that $uc \gg l_t$. Since $l_k \neq c$, then $c \gg l_t$. Since l_t is a Lyndon word, then we already have that for any suffix z of l_t, $z \gg l_t$. Hence, for any $f' < f$, Fact 1 yields $z(l_t)^{f'} rest(l_t)c^i \gg (l_t)^f rest(l_t)c^i$. In conclusion, wc is a Lyndon word, which contradicts the assumption that l_t is right-stable.

For the second part of the proof, we show that if $rest(l_t)$ is not a prefix of l_t, then l_t is right-stable.

Assume for the moment that l_t is not a prefix of $rest(l_t)$, i.e., neither of l_t or $rest(l_t)$ is a prefix of the other. Then, letting v be the longest common prefix of l_t and $rest(l_t)$, we have $|v| < \min\{|l_t|, |rest(l_t)|\}$. But then there are symbols a and a' such that $a \neq a'$, va is a prefix of l_t and va' is a prefix of $rest(l_t)$. Now, it is known from [10] that $a < a'$ implies that $l_t l_{t+1} ... va'$ is a Lyndon word. Hence, $a < a'$ is impossible in our case, since it would violate Fact 3. It is convenient to carry out this part of the proof explicitly, both for completeness of presentation and for future reference. Recall that, since l_t is a Lyndon word, then for every suffix z of l_t, we have $z \gg l_t$ and, by Fact 1, $z(l_t)^g va' \gg (l_t)^f va'$, where f was defined earlier in this proof and $g < f$ is a natural number. Thus, we only need to show that for every suffix z of va', it is also $z \gg l_t l_{t+1} ... va'$. This is obvious for $z = va'$, due to the hypothesis that va is a prefix of l_t and $a < a'$. Let then z be an arbitrary proper suffix of va', and write $z = z'a'$. Since v is a prefix of l_t, then clearly $z' \ll l_t$ is impossible, by Fact 2. Hence $z' < l_t$ implies that z' is a prefix of l_t. Thus, there exist a'' in Σ such that $z'a''$ is a prefix of l_t. But va is also prefix of l_t, and z' is a suffix of v. Therefore, it must be $a \geq a''$, by Fact 2. Hence, $a' > a$ yields $a' > a''$ and $z = z'a' \gg z'a''$, that is, $z \gg l_t va'$. In conclusion, assuming $a' > a$ yields that $l_t va'$ is a Lyndon word. This violates Fact 3. Thus, since $a \neq a'$, it must be that $a > a'$, whence the claim is established for l_t not a prefix of $rest(l_t)$.

We now show that l_t cannot be a prefix of $rest(l_t)$. In fact, let $g > 0$ and u be chosen, respectively, as the maximum integer and the longest prefix of l_t for which $(l_t)^g u$ is a prefix of $rest(l_t)$. Clearly, $rest(l_t) = (l_t)^g u$ is impossible, for otherwise every factor following l_t in the decomposition of x except u would be identical to l_t, and we would have $rest(l_t) = u$ and u a prefix of l_t, in contradiction with our assumptions. Thus, there are symbols a and a' such that $a \neq a'$, $(l_t)^g ua'$ is a prefix of $rest(l_t)$ and ua is a prefix of l_t. Assuming $a > a'$ yields that the first g factors in the decomposition of $rest(l_t)$ are each identical to

l_t, which contradicts the definition of $rest(l_t)$. Assuming $a < a'$ implies instead, through an argument already used earlier in this proof, that $l_t l_{t+1} ... ua'$ is a Lyndon word, which contradicts the assumption that l_t is a factor in the decomposition. Hence, l_t cannot be a prefix of $rest(l_t)$. •

4. COMBINATORICS OF COMPOSITIONS

In this section, we study how the decompositions of two strings x and x' are related to the decomposition of string xx'. This is easy in the case where both x and x' are Lyndon words, in view of the following known fact (cf., e.g., [10]).

Fact 4. Let u and v be Lyndon words. Then uv is a Lyndon word if and only if $u < v$.

In general, the composition of two factorizations is less straightforward. Of course, right-stable factors of x are not affected by the extension of x into xx', but we know that in the general case at least one factor is not right-stable. We start by listing two lemmas that shall be of use later.

Lemma 1. Let l and l' be two distinct Lyndon words such that, for some $u \in \Sigma^+$ and $v \in \Sigma^*$, we have that $l' = uv$, and u is a suffix of l. Then, lv is a Lyndon word.

Proof. The assertion trivially holds if v is empty, thus we assume henceforth that v is not empty. Since l is a Lyndon word, then for any suffix u' of l we have $u' \gg l$, whence, by Fact 1, also $u'v \gg lv$. Letting now v' be v or some suffix of v we have also $v' \gg l' > u \gg l$, which concludes the proof. •

Lemma 2. For $u \in \Sigma^+$ and $v \in \Sigma^*$, let $l = uv$ be a Lyndon word. Then, for any $w \in \Sigma^*$, $|v|$ is the position of a factor in the decomposition of vlw.

Proof. Let l' be the last factor in the decomposition of v. Then, l' is border-free. Let u be the longest prefix common to l' and l. Since l is a Lyndon word, we have $|u| < |l'|$, whence there are symbols a and a' such that ua' is a prefix of l' and ua is a prefix of l. Now $a' < a$ is impossible, since it violates Fact 2. Hence, it must be $a > a'$, i.e., l' is right stable in vl. The rest of the claim is an immediate consequence of Fact 3. •

From now on and until stated otherwise, $l'_1 l'_2 ... l'_{k'}$ is the factorization of a nonempty string x'. Consider the string xx', where x has factorization $l_1 l_2 ... l_k$, and let d be the

minimum index in the decomposition of x for which factor l_d is not right-stable. From now on, when referring to a string z, we use z as a subscript of *rest* and *tail*. Let $f \geq 1$ be the integer value for which $tail_x(l_{d-1}) = (l_d)^f rest_x(l_d)$. For an arbitrarily large m, let y be the longest prefix common to $(l_d)^m$ and $rest_x(l_d)x'$. We have, by Theorem 2, that $|y| \geq |rest_x(l_d)|$. Clearly, l_d is either a factor or the proper prefix of a factor in the decomposition of xx'.

Lemma 3. Assume that $|y| < 2|l_d|$, and let g be the largest index for which $s = rest_x(l_d)l'_1 l'_2 ... l'_g$ is a prefix of $(l_d)^2$. Then one of the following cases applies:

Case 1: each one of the consecutive occurrences of l_d in $tail_x(l_{d-1})x'$ is a factor in the decomposition of xx'; moreover, each such occurrence is right-stable in xx' iff $y \neq rest_x(l_d)x'$.

Case 2: the word $z = tail_x(l_{d-1})l'_1 l'_2 ... l'_{g+1} = (l_d)^f s l'_{g+1}$ is a Lyndon word.

Before starting with the proof, we observe that, since all the l_i's with $i < d$ are right-stable, then, in the light of Fact 3, we get that in Case 2 z is a factor or the prefix of a factor in the decomposition of xx'.

Proof. The assertion holds if we have $y = rest_x(l_d)x'$, since in this case the suffix of xx' at position $|l_1 l_2 ... l_{d-1}|$ is in the form $(l_d)^i l$ with $f \leq i \leq f+1$ and $l = l_d$ or l is a proper prefix of l_d. Hence, Case 1 of the claim applies, since l_d is a Lyndon word. Clearly, the occurrences of l_d in such a suffix are not right-stable factors for xx'.

Assume henceforth $y \neq rest_x(l_d)x'$. We have the following alternatives.

(A) $|y| \neq |l_d|$. This case encompasses two subcases depending on whether (subcase A1) $|y| < |l_d|$ or (subcase A2) $|l_d| < |y| < 2|l_d|$. Clearly, there exist two distinct symbols a and a' such that a' is a symbol of l'_{g+1}, ya is a prefix of l_d, and either (subcase A1) ya' is a prefix of $rest_x(l_d)x'$, or (subcase A2) $l_d ya'$ is a prefix of $rest_x(l_d)x'$.

Irrespective of which subcase applies, if $a > a'$, then we have Case 1, with every consecutive factor identical to l_d being also right-stable in xx'. For $a < a'$, we have that a prefix of $tail_x(l_{d-1})x'$ is either in the form $w = (l_d)^f ya'$ ($|y| < |l_d|$), or in the form $w = (l_d)^{f+1} ya'$ ($|l_d| < |y| < 2|l_d|$) In either subcase, w is a Lyndon word. Case 2 of the claim then follows by applying Lemma 1 to the Lyndon words w and l'_{g+1}.

(B) $|y| = |l_d|$. This is similar to the previous alternative: we have now $a \neq a'$, where a is the first symbol of l_d and a' is a symbol of l'_{g+1}. Clearly, $a < a'$ yields Case 2, while $a > a'$ yields Case 1, with every factor right-stable in xx'. •

Lemma 4. Assume that $|y| \geq |l_d|$, and that $l_1, l_2, ..., l_d$ are the first d factors in the

decomposition of xx'. Then either $p = |l_d| - |rest_x(l_d)| = |x'|$ or p is the position of a factor in the decomposition of x'.

Proof. The claim holds trivially for $p = |x'|$, so that we concentrate on the case $p < |x'|$. Assume first $|y| = |l_d|$ and let a and a' be defined as in alternative B of Lemma 3. Since l_d is a factor in the decomposition of xx', then Case 2 of Lemma 3 cannot apply, and we have $a > a'$. Assume that a' is not the first symbol of l'_{g+1}. Then there is a nonempty word u such that u is a prefix of l'_{g+1} and also a suffix of l_d. Let a'' be the first symbol of u. Since l'_{g+1} is a Lyndon word, we must have $a' \geq a''$. But then also $a > a''$, contradicting the assumption that l_d is a Lyndon word.

Let now $|y| > |l_d|$, and assume that p is not the position of a factor in the decomposition of x'. Then, there is a factor l' in such a decomposition such that $l' = u'v'$ with u' nonempty and u' a suffix of l_d. But then Lemma 1 ensures that $(l_d)^{f+1}v'$ is a Lyndon word, and thus a factor or the prefix of a factor in the decomposition of xx'. This contradicts the assumption that l_d is the d-th factor in such a decomposition. •

Lemma 5. Assume $|y| \geq 2|l_d|$, i.e., $(l_d)^2$ is a prefix of $rest_x(l_d)x'$, and that $l'_j = l_d$ is a factor in the decomposition of x', but $l'_i \neq l_d$ for $i < j$. Let t be the largest integer for which $(l_d)^t$ is a prefix of $tail_x(l_{d-1})x'$. Then, every occurrence of l_d in this prefix of $tail_x(l_{d-1})x'$ is a factor in the decomposition of xx'.

Proof. Since l_d is border-free, then l'_j must coincide with the second occurrence of l_d in $rest_x(l_d)x'$. Let g be the largest integer and $u \in \Sigma^*$ the longest prefix of l_d such that $(l_d)^g u$ is a prefix of $rest_x(l_d)x'$. Since l'_j is a factor in the decomposition of x', then either $tail'_x(l'_j) = (l_d)^{t-2}u$ or else $tail'_x(l'_j) = (l_d)^{g-2}uav$ for some $a \in \Sigma$ and $v \in \Sigma^*$ such that $ua \ll l'_j = l_d$. The assertion clearly holds in both cases. (Note that, in the second case, every occurrence of l_d is right-stable in xx'.) •

Lemma 6. Assume that $(l_d)^2$ is a prefix of $rest(l_d)x'$. Let l_j be defined as in Lemma 5, but assume that l_j is not a factor in the decomposition of x'. Then there is a nonempty prefix v of x' such that $tail_x(l_{d-1})v$ is a Lyndon word and $|v| \neq |x'|$ implies that $|v|$ is the position of a factor in the decomposition of x'.

Proof. We know from Lemma 2 and Fact 3 that l_j must be the prefix of a factor in the decomposition of x'. Let y be this factor. Since both l_d and y are Lyndon words and $l_d < y$, then repeated application of Fact 4 yields that $(l_d)^i y$ is also a Lyndon word for any $i \geq 1$. Obviously, $|l'_1 l'_2 ... l'_{j-1} y|$ is the position of a factor in the decomposition of x'. This establishes the claim. •

We say that the two strings x and x' have a *simple composition* if x is a Lyndon word. In the following theorem, we make the convention that $l'_{k+1} = l_0 = \lambda$. In informal terms, the theorem shows that the factorization of xx' can be always split into two segments with the following properties. The first segment is simply a prefix of the factorization of x. The second segment is the solution to a problem of simple composition that involves an identifiable Lyndon word and a suffix $\bar{x}'$ of x' such that the factorization of $\bar{x}'$ is a suffix of the factorization of x'.

Theorem 3. Assume that x and x' do not have a simple composition. Then, there are always integers m and i', with $m > 0$ and $1 \leq i' \leq k'+1$, and a nonempty Lyndon word l such that we can write $xx' = \bar{x}\bar{x}'$, with $\bar{x} = l_0 l_1 l_2 ... l_i (l)^m$, $\bar{x}' = l'_{i'} l'_{i'+1} ... l'_{k'} l'_{k'+1}$, and such that either xx' has decomposition $l_0 l_1 l_2 ... l_i (l)^m l'_{i'} l'_{i'+1} ... l'_{k'} l'_{k'+1}$ or else $m = 1$ and l and $\bar{x}'$ have a simple composition.

Proof. Let $d \leq k$ be the smallest index for which l_d is not right stable in x. We distinguish the following cases.

Case $l_d = l_k$.

The claim holds trivially if x' is of the form $(l_d)^c u$, with u a prefix of $l_d = l_k$ and $c \geq 0$. Assume next that, for some $c \geq 0$ and distinct symbols a and a', we have that $(l_d)^c ua$ is a prefix of $tail_x(l_{d-1})x' = (l_d)^{k-d+1}x'$ and $(l_d)^c ua'$ is a prefix of $(l_d)^{k-d}x'$. If $a > a'$, then we know that each one of the consecutive $c+1$ occurrences of l_d in $tail_x(l_{d-1})x'$ is a factor (in fact, a right-stable factor) in the decomposition of xx', and that l'_{c-k+d} is the first factor in the decomposition of u. Setting then $i = d-1$, $i' = m = c-k+d$ and $l = l_d$ clearly meets the claim. If $a < a'$, then $w = (l_d)^{c+1} ua'$ is a Lyndon word, and we know from the preceding lemmas that w is also the prefix of a Lyndon word z such that $|z| - (k-d+1)|l_d|$ is either $|x'|$ or the position of a factor l'_t in the decomposition of x'. Clearly, setting $i = d-1$, $l = z$, $m = 1$ and $i' = t$ satisfies the claim.

Case $l_d \neq l_k$.

By Theorem 2 and the definition of $rest$, we have that $rest_x(l_d)$ is a proper prefix of l_d. Since $l_d \neq l_k$, we also have that $rest_x(l_d) \neq \lambda$. Assume first that the condition of Lemma 3 is satisfied and results in an instance of Case 2. Then, taking $i = d-1$, $l = z$ and $i' = g+2$, where z and g are defined in that lemma, clearly meets the claim. The claim is easily met for each one of the cases considered in Lemmas 4-6, i.e., in all cases where $l_d = rest_x(l_d)v$ with v a prefix of x'. In brief, this is due to the fact that in all these cases there is one occurrence of l_d which originates in x and terminates in x'. Then, either l_d is a factor in the decomposition of xx' and we choose $l = l_d$ (Lemmas 2 and 4 guarantee

then the existence of i' as specified in the claim), or else l_d is the prefix of a Lyndon word that originates in x and ends in x' (then Lemma 6 guarantees the existence of i').

We are left now with the instances of $d < k$ where Case 1 of Lemma 3 occurs. The existence of i' is no longer guaranteed for the choice $l = l_d$. However, we know that under these conditions l_d and every subsequent replica of it are right-stable factors in xx'. Let l_t be the first factor in the decomposition of $rest_x(l_d)$. If $l_t = l_k$, then we have seen that it is possible to satisfy the claim. If $l_t \neq l_k$, then $rest_x(l_d)$ is not border-free. It is also easily seen that, for $l_t \neq l_k$, l_t cannot be right-stable (since $rest_x(l_d)$ is a prefix of l_d, then imposing that l_t be right-stable would violate Fact 2 for l_d). In summary, for $l_t \neq l_k$ we have that l_t is not right stable, and there is an integer $c \geq 1$ such that $tail_x(l_{t-1}) = (l_t)^c rest_x(l_t)$, with $rest_x(l_t)$ a nonempty prefix of l_t. Applying to l_t the case analysis previously developed for l_d either satisfies the claim or else yields that the c replicas of l_t in x are right-stable. Repeated application of this treatment yields the claim. •

Assume now that we are given a Lyndon word l and the decomposition $l_1, l_2, ..., l_k$ of some string x. At this point, we are interested in the relation between the decomposition of x and the decomposition $\bar{l}_1 \bar{l}_2 ... \bar{l}_k$ of lx. This is the case where we say that l and x admit of a simple composition, which seems to imply that the structure of the decomposition of lx is related in some trivial way to the structures of l and of the decomposition of x. This is not always the case. In fact, appending just one symbol to the left of some string x may upset the entire decomposition of x. For example, let $x = cbcbbcbcbbcbcabbc$. We have $l_1 = c, l_2 = bc, l_3 = l_4 = bbcbc, l_5 = abbc$. If we append an a to the left of x, we get, for ax, that $\bar{l}_1 = acbcbbcbcbbcbc$ and $\bar{l}_2 = abbc$. The following is an easy consequence of lemmas 5 and 6.

Lemma 7. Assume that Lyndon word l is a prefix of x. Then either $\bar{l}_1 = l$, or else $\bar{l}_1 = ll_1$.

Proof. That only one of the cases in the claim may apply follows from Fact 3. The rest of the claim is an easy corollary of lemmas 5 and 6. •

Theorem 4. Let t be the smallest index in the decomposition of x such that $w = ll_1 l_2 ... l_{t-1} \geq l_t$. Then, $\bar{l}_1 = w$ and $\bar{l}_2 = l_t$.

Proof. By our choice of t, we have that, for every $d < t$, $ll_1 l_2 ... l_d < l_{d+1}$. But then, using Fact 4, it is easy to establish by induction on d that that $ll_1 l_2 ... l_d$ is a Lyndon word for $d \leq t$. The assertion then follows from Lemma 7 in case of equality between w and l_t. Assume henceforth $l_t < w$. The assertion is obvious if $l_t \ll w$, thus we assume henceforth that l_t is a prefix of w. Let u be the longest common prefix of w and $l_t tail_x(l_t)$. The

claim holds clearly when $u = l_t tail_x(l_t)$. Assuming then $u \neq l_t tail_x(l_t)$, there are distinct symbols a and a' such that ua is a prefix of w and ua' is a prefix of $l_t tail_x(l_t)$. We show that $a < a'$ is impossible, thus establishing the claim also in this case. Since u is a prefix of a Lyndon word, then u can be written as $(vax)^m v$ for some integer m and Lyndon word vax. If $a > a'$, then $ua' = (vax)^m va'$ is also Lyndon word. But l_t is a proper prefix of ua', which contradicts Fact 3. •

5. ALGORITHMIC IMPLEMENTATION

In this section, we use the results developed earlier in the paper to design a CRCW PRAM algorithm for computing the Lyndon factorization of an input word s of $|s| = n$ symbols. We assume to be given n processors $p_1, p_2, ..., p_n$, that have simultaneous, random access to a memory bank consisting of $O(n \cdot T)$ locations, where T is the total time taken by our computation. We say that processor p_i $(i = 1, 2, ..., n)$ has *serial number* i. The input is stored into an array of consecutive locations of the common memory, and processor p_i is assigned to the i-th symbol of s $(i = 1, 2, ..., n)$. Any subset of the n processors can concurrently read from or write to the same memory location. When more than one processor attempts to write, we make the convention that the one with the smallest serial number succeeds. This variant of the model is called PRIORITY CRCW. In our application, this type of concurrent write can be simulated in constant time [11] by the weaker model where one processor at random succeeds in writing but it is not know in advance which one will succeed.

We adopt a standard divide-and-conquer scheme, consisting of $\log n$ stages each requiring constant time. Assuming w.l.o.g. that n is a power of 2, we regard the positions of the input string at the beginning of the S-th stage as partitioned into $n/2^{S-1}$ disjoint *blocks* each of size 2^{S-1}. Starting with the first block $[1, 2^{S-1}]$, we give all blocks consecutive ordinal numbers. For $S = 1, 2, ..., \log n$, stage S handles simultaneously and independently every pair formed by an odd-numbered block B and by the even-numbered successor B' of B. For every such pair, the goal of the stage is that of combining the already computed factorizations of the two substrings x and x' of s that are stored, respectively, into B and B' into the factorization of xx'. Thus, the main invariant is that at the beginning of stage S the factorization of every block of size 2^{S-1} has been computed. We call this *Invariant 0* . Invariant 0 trivially holds for $S = 1$, since the factorization of a single symbol is the symbol itself.

We only need to show how two blocks such as B and B' are combined. We apply to the two associated strings x and x' the notational conventions made in connection with Theorem 3. We need a few additonal notions, that are given next.

The first position of each block of s is called the *block head*, and the processor assigned to the head of a block is the *block representative* of that block. Since the block partitions are rigidly defined for each stage, then the position of any block head can be computed by any processor in constant time. Similarly, if l is a factor in the decomposition, say, of x, then the first position of l is the *head* of that factor, and the processor assigned to the first symbol of l is the *factor representative* of l. With respect to the decomposition of either x or x', an *l-run* is a maximal sequence of factors identical to l in such a decomposition. The total number of replicas of l in the l-run is the *size* of that run. For every run, the factor of the run having minimum index in its associated decomposition is called the *head* of that run; the representative of that factor is also the *run representative* of that run. Our scheme will maintain, in addition to Invariant 0, the following auxiliary invariants.

Invariant 1. If processor p is assigned to a position of factor l in the decomposition of x (respectively, x'), then p knows the address in B (resp., B') of the head of l as well as the address (or serial number) of the representative of the run containing l.

Invariant 2. The representative of an l-run knows $|l|$ and the size of that run.

The following three steps have the effect of combining the decompositions of x and x' into the decomposition of xx'. The first two steps take place only if x and x' do not have a simple composition. The combination of a single Lyndon word with the decomposition of a given string is instead the task of the third step. The condition that the factorization of x contains only one factor is easily checked in constant time, e.g., using two consecutive appropriate concurrent writes to the block head of B to see whether there are two distinct factor representatives. Henceforth, we assume that there are at least two factors in the decomposition of x. In the course of our description, we will say often that our scheme identifies or handles the *index* (i.e., ordinal number) of a factor in the decomposition of either x or x'. This phraseology is used only in order to relate in a clearer way to the results of the previous section. It should be understood that, in actuality, the scheme only identifies and handles the heads of the factors having the said indices. This distinction is important, since our time bound would not be achieved if an explicit computation of factor indices had to take place at each stage.

Step 1. The goal of Step 1 is to detect the factors of x that are not right-stable. Theorem 2 is the handle for this. For every factor y of x, the representative of the y-run computes, using Invariant 2, the position of $rest_x(y)$ in B. If $|rest_x(y)| > |y|$, then (every factor in the y-run is right-stable) the run representative simply sets an appropriate flag in the first position of the run head. If, on the other hand, $|rest_x(y)| \leq |y|$, then the $|y|$ processors assigned to the head of the run inspect the first $|y|$ symbols of $rest_x(y)x'$.

Specifically, the processor assigned to the d-th symbol of y ($d = 1, 2, ...|y|$) checks whether that symbol matches the d-th symbol of $rest_x(y)x'$. Note that this processor can actually compute the value d using Invariant 1. Subsequently, all processors detecting a mismatch attempt to write their respective serial numbers into a memory cell uniquely associated with the representative of the run. In our working model, the processor having smallest index succeeds. The representative of the run can now check in constant time whether the right-stability condition of Theorem 2 is satisfied, in which case it sets the flag located in the first position of the the run head. Using Invariant 1 again, every processor of block B can learn at this point by inspection of the appropriate flag whether the factor it is assigned to is right-stable or not. The processors assigned to right-stable factors will remain idle for the remainder of the stage. All others (recall that there is always at least one non right-stable factor) proceed to Step 2.

Step 2. The main goal of this step is to identify i, i', l, m and the position in x' of $\bar{x}'$, as per Theorem 3. If l and its possible subsequent replicas are found to be factors in the decomposition of xx', then such a decomposition will also be computed in Step 2, whence Step 2 will be terminal for the stage. If this is not the case, then Theorem 3 tells us that $m = 1$ and l and $\bar{x}'$ have a simple composition. In conclusion, if Step 2 is not terminal for the stage, we will only have to solve a well defined problem of simple composition, and this problem will be handled in Step 3. The details of Step 2 cannot consist of a mere recapitulation of the criterion set forth in Theorem 3. In fact, we are only allowed constant time for the stage, which forbids pursuing the cascaded tests in that theorem. Our approach will be instead to exploit the constant time min computation inherent to our working model in order to reach quickly the bottom of the iterated argument subtending Theorem 3. From that point on, our main concern will be to show how the n processors can exchange information efficiently and carry out the rest of the work in constant time.

The opening action of Step 2 is the following test: for each run of non-right-stable factors in the decomposition of x, the processors assigned to the head l_d of the run test simultaneously and in constant time whether $l_d < rest_x(l_d)x'$. The details of this test are similar to those of Step 1. If no run head passes the test, then by Theorem 3 every factor in the decomposition of x is right stable in xx'. (The condition $l_d = rest_x(l_d)x'$ is impossible at this point, since we ruled out that x is a Lyndon word.) Hence, the decomposition of xx' consists simply of the concatenation of the decomposition of x and that of x'. The operation of the stage is complete, since Invariant 0 holds now for xx' and invariants 1 and 2 are trivially propagated to next stage.

Assume, on the other hand, that at least one run head passes the test. In this case, all run representatives passing the test use common-write to the block head in order to identify,

in constant time, the successful run head having smallest index in the decomposition of x. Let l_t be the winning run head. By Theorem 3, every factor preceding l_t in the decomposition of x is now right-stable in xx', while l_t is either a factor or the prefix of a factor in the decomposition of xx'. The remainder of the stage takes one of two possible avenues, according to whether or not the condition $rest_x(l_t) = \lambda$ is satisfied.

Alternative 1: $rest_x(l_t) = \lambda$. This splits into two subcases according to whether or not l_t is a prefix of $rest_x(l_t)x'$.

Assume first that l_t is a prefix of x'. Then (cf. Fact 3) l_t either is identical to or is the prefix of the first factor l'_1 in the decomposition of x'. Which case applies can be learned by inspecting the factor representative of factor l'_1, which is also the run representative of the l'_1-run in the decomposition of x'. If l'_1 is identical to l_t, then l'_1 must surrender its status of run head to the head of the l_t-run. Each one of the factor representatives of a factor formerly in the l'_1 run learns about this change by inspection of their old run representative, and updates its knowledge of the run representative accordingly. This preserves invariants 1 and 2. Having accomplished the decomposition of xx' the stage terminates. If $|l'_1| > |l_t|$, then l_t and all its subsequent replicas in x must coalesce with l'_1 into a single Lyndon word. In the most general case, l'_1 will be the head of a run of size larger than 1. Before combining with $tail_x(l_{t-1})$, l'_1 must pass its leadership on to the second factor in this l'_1-run. This is easily accomplished in constant time, due to invariants 1 and 2. The processors now assemble the Lyndon word $z = l_t tail_x(l_t) l'_1$ (cf. Case 2 of Lemma 3) and enter Step 3 with $i = t-1, m = 1, l = z$ and i' the index of the first factor following l'_1 in x'.

Assume now that l_t is not a prefix of x'. We need to explain how the word z (that will serve as the parameter l to be passed on to Step 3) is computed. With the notation of Lemma 3, the main problem is that of identifying l'_{g+1}. Let ya' be the shortest prefix of $rest(l_t)x'$ that is not a prefix of l_t. The l_t-run representative p identifies the factor head of the factor of x' containing a' (this factor is called l'_{g+1} in Lemma 3). Processor p achieves this simply by inspecting the information about the factor head stored in the processor assigned to a'. At an extra constant time, p can also identify the run head of the l'_{g+1}-run containing l'_{g+1}. Again, l'_{g+1} or its run-representative may have to surrender the status of run-representative to the successor l'_{g+2} of l'_{g+1}, if $l'_{g+1} = l'_{g+2}$, and this is done as earlier using the invariants. Once this is done, the l_t-run representative can assemble word z of Lemma 3, using its own information and that stored in the old head of the l'_{g+1}-run. Every processor assigned to a position of z updates its pointers to both factor- and run representative, so that both point now to the old run representative of the l_t-run. The procedure now enters Step 3 with $i = t-1, i' = g+2, l = z$ and $m = 1$.

Alternative 2: $rest_x(l_t) \neq \lambda$. This case splits into two subcases depending on whether

or not $(l_t)^2$ is a prefix of $rest_x(l_t)x'$. In the following, we use v to denote the prefix of x' of length $|l_t| - |rest_x(l_t)|$.

Assume first that $(l_t)^2$ is a prefix of $rest_x(l_t)x'$. Then Lemma 2 guarantees that $|v|$ is the position of a factor in the decomposition of x'. In other words, the decomposition of the suffix $\hat{x}'$ of x' that is obtained by deleting the first $|v|$ symbols of x' is a suffix of the decomposition of x'. Consider now the word xv. Clearly, the decomposition of xv can be obtained by that of x by first deleting all factors in the decomposition of $rest_x(l_t)$ and then appending a new factor identical to l_t, i.e., the factor contributed by the Lyndon word $rest_x(l_t)v$. At this point, the problem of composing the decompositions of xv and $\hat{x}'$ is identical to the problem handled in the first subcase of Alternative 1. In the light of the preceding discussion, the manipulations that lead the processors to extend the decomposition of x into that of xv, and truncate x' into $\hat{x}'$, are trivial and are omitted.

Finally, assume that $(l_t)^2$ is not a prefix of $rest_x(l_t)x'$. If $|v|$ is the starting position of a factor in the decomposition of x', then this is similar to the subcase of Alternative 2 that was just discussed. Observe that, under our current assumptions, if l_t is to stay as a factor in the decomposition of xx' then $|v|$ is guaranteed to be the position of a factor in the decomposition of x', by virtue of Lemma 4. All other cases of Alternative 2 are similar to the second subcase of Alternative 1.

Step 3. If Step 3 is entered, then we have identified a prefix $\check{x}$ of x such that the decomposition of $\check{x}$ is a prefix of the decomposition of x and also a prefix of the decomposition of xx'. We also have a Lyndon word z and a suffix $\bar{x}'$ of x' such that we know the decomposition of $\bar{x}'$ and we know that $xx' = \check{x}z\bar{x}'$. Thus, solving the problem of simple composition for z and $\bar{x}'$ is all is needed to complete the stage. Step 3 consists of applying first Lemma 7 and then possibly Theorem 4 to the arguments $l = z$ and $x = \bar{x}'$. If the test implied by Lemma 7 fails, the comparisons of Theorem 4 are carried out simultaneously by the processors assigned to the factors of $\bar{x}'$. The block head of B is used to identify the position of z in xx'. The remaining details are trivial at this point and are thus omitted.

This concludes the description of our scheme. The discussion of this section establishes the following result.

Theorem 5. The Lyndon decomposition of a word of n symbols can be computed by a CRCW PRAM with n processors in time $O(\log n)$ and linear space.

6. APPLICATIONS

1 - The minimum suffix of all prefixes of a string.

The most immediate application of the parallel algorithm of Section 5 is to the computation of the minimum suffix of the input string. In fact, the minimum suffix of a string is precisely the last factor in the Lyndon decomposition of that string [10]. Just as it happens for its serial predecessor [10], some light upgrades of our algorithm lead to actually compute the minimum suffix for *every* prefix of the string, leaving time and processor bounds unaltered. (We note, incidentally, that computing the minimum suffixes of all prefixes of a string yields an implicit description of the Lyndon decompositions of all such prefixes.)

To start the discussion of our method, we single out in Fact 5 below a useful characterization due to Duval [10] of the class $\mathcal{P}$ of all words that are nonempty prefixes of Lyndon words. A proof of Fact 5 is also pespersed in the proof of Theorem 2. Let $\mathcal{S}$ be the set of all words in the form $(uv)^k u$, where $u \in \Sigma^*$, $v \in \Sigma^+$, $k \geq 1$ and uv is a Lyndon word.

Fact 5. $\mathcal{P} = \mathcal{S}$ if the alphabet Σ is unbounded, and $\mathcal{P} = \mathcal{S} - \{c^k | k \geq 2\}$, where c is the maximum symbol in Σ, otherwise.

We assume familiarity of the reader with the notion of *period* of a string. We say that a period q of a word w is *nontrivial* if $q \neq |w|$.

Lemma 8. Let w be a word in $\mathcal{P}$, and set $p = 0$ if w is unbordered, and p equal to the maximum nontrivial period of w otherwise. Then p is the position in w of the last factor in the Lyndon decomposition of w.

Proof. By Fact 5, we can write w in the form $(uv)^k u$, where $u \in \Sigma^*$, $v \in \Sigma^+$, uv is a Lyndon word, and either u is not empty and $k \geq 1$ and or u is empty and $k > 1$. If u is empty, then the decomposition of w consists of k factors all identical to uv, and the position of the last factor is $(k-1)|uv|$. We also have $p = (k-1)|uv|$, since uv is unbordered. For u not empty, $k|uv|$ is a nontrivial period of w. The claim thus holds if $p = k|uv|$. If, on the other hand, $p \neq k|uv|$, then it must be $p > k|uv|$, i.e., u is bordered. Since $u \in \mathcal{P}$, we can apply to u the analysis previously applied to uv. Iteration of this argument yields the claim. •

Let $l_1 l_2 ... l_k$ be the decomposition of some word x. Combined with Fact 3, Lemma 8 shows that the minimum suffixes of all prefixes of x can be obtained by computing the minimum suffixes of all prefixes of each individual factor in the decomposition of x. From now on, we can therefore assume that the Lyndon decomposition at the outset of the algorithm of the previous section is given, and concentrate on a single factor l in such a

decomposition. The understanding is that the manipulations performed on l take place synchronously on all other factors of the decomposition of x.

In view of Lemma 8, it is not surprising that $|l|$ processors can compute all mimimum suffixes of Lyndon word l in $O(\log|l|)$ time, due to the strong relationship between this problem and plain string searching. Specifically, consider the table $reach_l$ such that $reach_l[i]$ equals the maximum length of a prefix of l that occurs also at position i ($i = 0, 1, 2, ..., |l|-1$). Given an integer $k \leq i$, we say that the i-th symbol of l is *dominated* by $reach_l[k]$, iff $k + reach[k] \geq i$. For $i = 1, 2, ..., |l|$, define now $close_l[i]$ as the largest k such that $reach_l[k]$ dominates $l[i]$. At this point, the criterion of Lemma 8 translates into saying that, for every prefix $l[1]l[2]...l[i]$ of l, the position in l of the minimum suffix of $l[1]l[2]...l[i]$ is $close_l[i]$. It is not difficult to compute either one of the tables $reach_l$ or $close_l$ with $|l|$ processors in $O(\log|l|)$ time, e.g., by adaptation of the string searching algorithm in [12]. Thus, given the Lyndon decomposition of a string x, the minimum suffixes of all prefixes of x can be computed by a CRCW PRAM with n processors in $O(\log n)$ time and linear space.

2 - The maximum suffixes of all prefixes of a string.

As pointed out in [10], this problem is not symmetric to the previous one. However, it is known that the lexicographically maximum suffix of a string x with respect to the converse of the order relation "<" is the longest one among the suffixes of x that belong also to the set $\mathcal{S}$ defined earlier. We will use a re-statement of this property to compute the maximum suffixes of all prefixes of x in $\log n$ CRCW steps, with n processors.

Let l be a factor in the decomposition of x and i the position of l in x. We say that an integer j is *covered* by l if either $(j - i) \leq |l|$ or $rest_{x[1]x[2]...x[j]}(l) = tail_{x[1]x[2]...x[j]}(l)$ and $rest_{x[1]x[2]...x[j]}(l)$ is a prefix of l. Let $\mathcal{C}$ be the set of all factors covering j, and $l_m(j)$ the element of $\mathcal{C}$ having minimum index in the decomposition of x. The following property yields a criterion for finding all maximum suffixes of x.

Fact 6. The position in x of the maximum suffix of $x[1]x[2]...x[j]$ equals the position in x of the run head of the $l_m(j)$-run.

We refer to [10] for a justification of Fact 6. Given the Lyndon decomposition of x in the format specified in Section 5, it is easy to compute, for all j's, the positions of $l_m(j)$ and its run head, using n processors and $\log n$ steps. To avoid many tedious details, we describe the method informally. Let $l^{(1)}(j)$, $l^{(2)}(j)$, ..., $l^{(h)}(j)$ be the factors of x covering j. For any d in $[1, h]$, aligning factor $l^{(d)}(j)$ with $tail_x(l^{(d)}(j))$ will bring onto position j precisely one among the processors assigned to $l^{(d)}(j)$. Note that, for any j, there is a group of processors uniquely assigned to j in this way. Clearly, all processors assigned to

j can perform a binary search driven by the length of x to identify the minimum among their own serial numbers. The j-th position of an array, serving as the target for common writes, can be used to perform the binary search. Having found the minimum, invariants 1-2 lead to the identification of $l_m[j]$ and, from there, to the head of the $l_m(j)$-run.

9 - The lexicographically least rotations of all prefixes of a string.

Given a word $x = x[1]x[2]...x[n]$, the i-th *rotation* of x ($i = 1, 2, ..., n$) is the string $w = x[i]x[i+1]...x[n]\ x[1]\ x[2]\ ...\ x[i-1]$. A *least lexicographic rotation (llr)* of string x is a rotation of x that is lexicographically smallest among all rotations of x. Since all rotations of x have equal length, then for any two such rotations w and w', $w \neq w'$ implies that w and w' differ in at least one symbol. An llr of x is completely identified by its starting position (mod. $|x|$) in xx. We call such a position a *least starting position (lsp)*. An lsp of x can be computed in linear time by serial computation. The fastest solution known was given in [14]. As pointed out in [10], the Lyndon decomposition of word xx will also expose an llr of x. This is due to the fact that the llr of x is either a Lyndon word or a power of a Lyndon word, and either case manifests itself while decomposing xx.

It is not difficult to compute the least rotation of x on a CRCW PRAM with n processors in $O(\log n)$ time. One possible approach is to perform, on xx, $\log n$ constant-time iterations of the following kind: At the i-th iteration, x is partitioned into $n/2^i$ blocks of size 2^i, and, for each block, we know the starting position of a lexicographic minimum among all substrings of length 2^{i+1} of x originating in that block. The iteration consists of combining pairwise the blocks and computing one minimum substring of size 2^{i+2} for each combined block. We clearly have enough processors to perform all substring comparisons in constant time. The only difficulty is when both candidate substrings from two combining blocks extend into identical minima. Using an observation already in [14], however, it is possible to always rule out one of the candidates in constant time, whence the overall computation is done in time $O(\log n)$.

Computing the llr's of all prefixes of x within the same bounds is more involved. For this task, we resort to a criterion recently established in [3], and used to compute the llr of all prefixes of x in overall linear time. Let l be one of the factors in the Lyndon decomposition of x. Define $prev(l)$ as the prefix of x that precedes the first occurrence of l. We say that l is a *special factor* of x if and only if $rest(l)$ is a prefix of l and, in addition, one of the following conditions is satisfied:

- $rest(l)$ is empty;
- l is a prefix of $rest(l)prev(l)$; or
- $l < rest(l)prev(l)$ but l is not a prefix of $rest(l)prev(l)$.

Observe that, for any word x, the Lyndon decomposition $l_1 l_2 l_k$ of x has at least one special factor, namely, l_k. As shown in [3], the following fact holds.

Fact 7. Let $l_1 l_2 ... l_k$ be the Lyndon factorization of a non-empty word x. Let t be the smallest index such that l_t is a special factor of x. Then $l_t ... l_k l_1 ... l_{t-1}$ is an llr of x, and $|prev(l_t)|$ is an lsp for x.

To see how Fact 7 can be used in our computation, assume that the table $reach_x$ has been computed. (Recall that the n processors can compute $reach_x$ in $O(\log n)$ time.) Consider now an integer $j \leq |x| = n$, and let as earlier $l_m(j)$ be the factor in the decomposition of x such that j is covered by $l_m(j)$ and m is minimum. Let i be the position of $l_m(j)$ in x and assume for generality that $(j-i) > |l_m(j)|$ (i.e., j does not fall inside $l_m(j)$). It is not difficult to show that $l_m(j)$ is a factor also in the decomposition of $x[1]x[2]...x[j]$. Once $reach_x$ and the position i of $l_m(j)$ in x are available, it takes constant time for processor p_j to test whether $l_m(j)$ (whence also the run head of the $l_m(j)$-run) is a special factor in the decomposition of $x[1]x[2]...x[j]$: This processor simply checks on $reach_x$ whether $j - |l_m(j)|$ is the starting position of a sufficiently long or (lexicographically) sufficiently small prefix of x. If the run head of the $l_m(j)$-run is a special factor for $x[1]x[2]...x[j]$, then, by Fact 7, the position of such a run head is also an lsp for $x[1]x[2]...x[j]$. Assume now that the test preformed by p_j fails, and, setting $f = j - |l_m(j)|$, consider the prefix $w = x[i+1]x[i+2]...x[f]$ of $l_m(j)$. Let k be the maximum length of a border of w. It is easy to see that the the factor following $l_m(j)$ in the decomposition of $x[1]x[2]...x[j]$ is $x[i+|l|+1]x[i+|l|+2]...x[j-k]$. Therefore, if $k = 0$, then $x[i+|l|+1]x[i+|l|+2]...x[j]$ is the last factor in the decomposition of $x[1]x[2]...x[j]$ and, also the earliest special factor in such a decomposition. Otherwise, the next Lyndon word to be tested by p_j as a special factor in the decomposition of $x[1]x[2]...x[j]$ is $x[i+|l|+1]x[i+|l|+2]...x[j-k]$. Note that p_j needs only to know k and $|l|$ in order to identify this word. If also $x[i+|l|+1]x[i+|l|+2]...x[j-k]$ fails the test, then we consider its rest in the decomposition of $x[1]x[2]...x[j]$ and apply the same treatment to it. In conclusion, given $reach_x$ and a mechanism for identifying the words to be considered in succession, it takes p_j time proportional to the number of words tested in order to compute an lsp for $x[1]x[2]...x[j]$. As is easily seen, the words considered by p_j are all replicas of shorter and shorter prefixes of $l_m(j)$, and they can be identified in succession by repeated application of the function "longest border of" to such prefixes. The "longest border of" function is actually the *failure function* [1] for $l_m(j)$, and it is not difficult to show that p_j will go through at most $\log |l_m(j)|$ applications of this function during its tests. Computing the failure function for a factor l is somewhat dual to the computation of the table $close_l$ discussed earlier, and is done easily in $O(\log |l|)$ time either starting from the table $reach_l$ or by direct adaptation of the techniques in [12]. In conclusion, n processors can compute the least rotations of all prefixes of a string in $O(\log n)$ time.

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman, "*The design and analysis of computer algorithms*", Addison-Wesley, 1974.

[2] Apostolico, A., M.J. Atallah, L.L. Larmore and H.S. McFaddin, "Efficient Parallel Algorithms for String Editing and Related Problems", *Proceedings of the 26-th Allerton Conference on Communications, Control and Computing*, Monticello, Ill. (Sept. 1988). Also, *SIAM Journal on Computing, to appear.*

[3] Apostolico, A. and M. Crochemore, "Optimal Canonization of All Substrings of a String", Purdue University CS TR 903 (1989). Also, *Information and Computation, to appear.*

[4] Apostolico, A. and Z. Galil (eds.), *Combinatorial Algorithms on Words*, Springer-Verlag Nato ASI Series F, Vol. 12, 1985.

[5] Apostolico, A., C. Iliopoulos, G. Landau, B. Schieber and U. Vishkin, "Parallel Construction of a Suffix Tree, with Applications", *Algorithmica* **3**, 347-365 (1988) .

[6] Berkman, O., D. Breslauer, Z. Galil, B. Schieber and U. Vishkin, "Highly Parallelizable Problems", *Proc. 21-st ACM Symp. on Theory of Computing*, Seattle, Wash., (May 1989), 309-319.

[7] Beame, P. and J. Hastad, "Optimal Bounds for Decision Problems on the CRCW PRAM", *Journal of the ACM 36*, **3**, 643-670 (1989).

[8] Chen, K.T., R.H. Fox and R.C. Lyndon, "Free Differential Calculus, IV", *Ann. of Math.* **68**, 81-95 (1958).

[9] Crochemore, M. and W. Rytter, "Usefulness of the Karp-Miller-Rosenberg Algorithm in Parallel Computations on Strings and Arrays", typescript, (1989).

[10] Duval, J.P., "Factorizing Words over an Ordered Alphabet", *Journal of Algorithms* **4**, 363-381 (1983).

[11] Fich, F.E., R. L. Ragde and A. Wigderson, "Relations between Concurrent-write Models of Parallel Computation", *Proceedings of the 3-rd ACM Symposioum on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 27-29), ACM, New York, 179-184 (1984).

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

[13] Lothaire, M., *Combinatorics on Words*, Addison Wesley, Reading, Mass., 1982.

[14] Shiloach, Y., "Fast Canonization of Circular Strings", *Journal of Algorithms* **2**, 107-121 (1981).