Workshop 06

Parallel Discrete Algorithms

A Simple Parallel Dictionary Matching Algorithm*

Paolo Ferragina

Dipartimento di Informatica
Università di Pisa, Italy. E-mail: ferragin@di.unipi.it

Abstract. In the Parallel Dictionary Matching problem a set of patterns $\mathcal{D}$ is fixed at the beginning, and the following Query(T) operation has to be quickly supported: given an arbitrary text $T[1:t]$, for each position i retrieve the longest pattern in $\mathcal{D}$ that is prefix of text suffix $T[i:t]$. In this paper, we present a simple CRCW PRAM algorithm achieving optimal work for answering Query(T) in the case of a constant-sized alphabet.

1 Introduction

The classical *pattern matching* problem on strings consists of finding all occurrences of a single pattern $P[1:p]$ as a substring of a text $T[1:t]$, where P and T are drawn from an ordered alphabet Σ. The goal is to preprocess P such that all the succeeding queries on an arbitrary text T can be answered quickly, that is, in optimal $O(t)$ time [7, 15]. The "dual" version of this problem, in which P is given on-line and T is fixed, has been studied as well attaining optimal solutions [16, 19].

One generalization of the pattern matching problem is the *multiple pattern matching* problem, commonly called *dictionary matching* (shortly, DM) problem. Here, instead of a single pattern, a *set* of patterns $\mathcal{D} = \{P_1, \ldots, P_k\}$, called the *dictionary*, is given to be preprocessed and an arbitrary text T is provided on-line with the intention of finding all the occurrences of the patterns in $\mathcal{D}$ that appear in T (let *tocc* be their number). In addition to its theoretical importance, DM problem has many practical applications. For example, in molecular biology, one is often concerned with determining the sequence of DNA, and then compare that sequence against all the known strings to find the ones that are related to it. Also, in computer virus detection applications, a dictionary of computer viruses is given and new programs are queried on-line to find out if they are *infected*.

Any pattern matching algorithm can be trivially extended to a set of patterns by matching each pattern separately, thus requiring $O(d + tk)$ time, where $d = \sum_{i=1}^{k} |P_i|$ is the dictionary size (i.e., brute-force method). However, one may clearly hope that, once $\mathcal{D}$ has been preprocessed, the cost of finding all the occurrences of $\mathcal{D}$'s patterns in T be proportional only to the length t of T and to the number *tocc*, independent of the length d of the (usually) much larger dictionary. Aho and Corasick [1] were the first to solve optimally the DM problem in $O(d \log \sigma)$ sequential time for the preprocessing of $\mathcal{D}$, and $O(t \log \sigma + tocc)$ time for answering a query on text T, where $\sigma = \min\{d, |\Sigma|\}$. This result is perhaps surprising because the text scanning time is *independent* of the dictionary size (for a constant-sized Σ). Since then, a dynamic formulation of this problem has been also well studied achieving very interesting results (e.g., see [2, 3, 4, 11]).

The DM problem has been also deeply investigated in the widely used Parallel Random Access Machine (shortly PRAM [13]). In particular, the powerful Concurrent-Read-Concurrent-Write variant of this model has been employed to describe various parallel solu-

* Work supported in part by MURST of Italy.

tions. We remark that in the parallel context, Query(T) operation is defined as follows: For each position of T, retrieve the *longest pattern* in $\mathcal{D}$ that occurs in T starting at that position. Notice that the whole information about shorter patterns is contained implicitly in this representation. Moreover, the output size does not prevent the algorithm to have polylogarithmic time complexity when using $O(t)$ processors. Amir and Farach [2] were the first to provide an efficient parallel algorithm for the DM problem requiring $O(\log m \log d)$ time and $O(t \log m \log d)$ work [2] for answering Query(T), and $O(\log m \log d)$ time and $O(d \log d)$ work for preprocessing $\mathcal{D}$, where m denotes the length of the longest pattern in $\mathcal{D}$. Then, Muthukrishnan and Palem [17] presented an algorithm requiring $O(\log m)$ time and $O(t \log m)$ work for answering Query(T), and $O(\log m)$ time and $O(d)$ optimal work for preprocessing $\mathcal{D}$. They also presented an improved algorithm, in the case of a constant-sized Σ, which requires $O(\log m)$ time and $O(d \log m)$ work for preprocessing $\mathcal{D}$, and $O(\log m)$ time and $O(t)$ optimal work for answering Query(T). Both two solutions use a large amount of space, i.e. $O(md^{1+\epsilon})$, for any given $\epsilon > 0$. Using *randomization*, first Amir and Farach and Matias [5], and later Farach and Muthukrishnan [9], have reported very efficient algorithms with expected work optimal bounds both for answering Query(T) and preprocessing $\mathcal{D}$.

In this paper we provide a *simple* CRCW PRAM algorithm achieving optimal work for answering Query(T) and requiring small space, in the case of a constant-sized alphabet. The dictionary can be preprocessed in $O(\log d)$ time and $O(d \log m)$ total work. Answering Query(T) requires $O(\log m)$ time and $O(t)$ optimal work. The total required space is $O(d^2 \log m)$. It is worth noting that our solution achieves the same time and work bounds as in [17], but it is simpler and also requires less space for $d^{1-\epsilon} = O(\frac{m}{\log m})$.

2 Preliminaries

In what follows we will use the classical naming technique [14] and the suffix tree data structure [16, 19] as basic tools to develop our parallel solution (see the corresponding literature for more details).

Let X be a string of x characters and assume $\$ \notin \Sigma$. [3] The suffix tree ST_X built on $X\$$ is a digital search tree containing all the suffixes of $X\$$ and occupying optimal $O(x)$ space [16]. The character $\$$ is used to prevent that a suffix $X[i : x]\$$ is a prefix of another suffix $X[j : x]\$$; thus there exists a unique leaf in ST_X for each suffix of $X\$$. Each arc of ST_X is labeled with a substring $X[i : j]$, which is represented as a triple (X, i, j). Given the suffix tree ST_X and a node u, we denote by $W(u)$ the concatenation of the labels on the path from the root to node u. Clearly, $W(u)$ is a substring of X and thus every ancestor w of u in ST_X denotes a string $W(w)$ which is a proper prefix of $W(u)$. In general, given a substring V of $X\$$, we define the (exact) *locus* of V as the node v in ST_X such that $V = W(v)$. Moreover, we define the *extended locus* of a string U as the node u in ST_X such that U is a prefix of $W(u)$ and $W(p(u))$ is a *proper* prefix of U, where $p(u)$ is the parent of u in ST_X.

The parallel construction of the suffix tree works on the arbitrary CRCW PRAM and requires two phases [6] (see [12] for a work-optimal algorithm). In the first phase, called *naming*, we label all of X's substrings of power-of-two length. Labels are integers between 1 and $x + 1$, and equal substrings get the same label (this is called *consistent naming*). In the second phase, called *refining*, a sequence of refinement trees $RT^{(r)}$ is produced for $r = \lceil \log x \rceil, \ldots, 0$. The final tree $RT^{(0)}$ is basically the suffix tree ST_X, except for some minor adjustments. For each intermediate value r, $RT^{(r)}$ is a better and better approximation of suffix tree ST_X.

[2] By *work* of a parallel algorithm, we mean the total number of operations performed to solve a problem [13]. A parallel algorithm is called *work optimal* if its work is of the order of time of the best possible sequential algorithm for the same problem.

[3] From now on we assume that alphabet Σ has constant size.

Theorem 1. *[6] Given a string $X[1:x]$, the names of all its substrings of power-of-two length and its set of refinement trees can be computed in $O(\log x)$ time and $O(x \log x)$ work. The total required space is $O(x^2 \log x)$.*

To search for the *longest prefix* of a string $Y[1:y]$ which occurs in string X, we maintain all of X's refinement trees and partition Y in substrings $\sigma_1, \sigma_2, \ldots, \sigma_k$, where $k \leq \lfloor \log y \rfloor, |\sigma_i| = 2^{r_i}, r_i > r_{i+1}$. Then, we label these substrings consistently with X and search for them in X's refinement trees, thus obtaining:

Lemma 2. *[6] Given an arbitrary string $Y[1:y]$, Y's longest prefix occurring in X (and its extended locus in ST_X) can be found in $O(\log y)$ time and $O(y)$ work.*

Before concluding this section, let us recall a simple result which will be used later.

Lemma 3. *[10] Let T be a tree in which the root and some nodes are marked. The pointer to the deepest marked ancestor of each node in T can be computed in $O(\log |T|)$ time and $O(|T|)$ optimal work on the EREW PRAM.*

3 Preprocessing $\mathcal{D}$

Let $\mathcal{D} = \{P_1, \ldots, P_k\}$ be a dictionary of *patterns* of total size $d = \sum_{i=1}^{k} |P_i|$ and maximal pattern length $m = \max\{|P_i| : 1 \leq i \leq k\}$ (w.l.o.g. assume m is a power of two). Since the dictionary is fixed at the beginning, our goal is to preprocess it by building a proper set of data structures to support work-optimal queries on arbitrary texts that are provided on-line.

Preprocessing $\mathcal{D}$ consists of two main steps. In Step (1), all the patterns in $\mathcal{D}$ are labeled and the corresponding set of refinement trees is built. In Step (2), the suffix tree ST_D, built on the patterns of $\mathcal{D}$, is augmented with some additional information.

Step (1): Consider the string $D = P_1\$_1 P_2\$_2 \ldots P_k\$_k$, where $\$_i \neq \$_j$ for each $i \neq j$, and $\$_i \notin \Sigma$ for all $i = 1, \ldots, k$. Notice that D's total length is still $O(d)$. Apply the naming technique to consistently label all of D's substrings having power-of-two length at most m (Theorem 1). Then, build the suffix tree ST_D and its set of refinement trees $RT^{(i)}$ only for $i = \log m, \ldots, 1, 0$, by exploiting the fact that all of D's substrings longer than m are distinct, so taking $O(\log m)$ time and $O(d \log m)$ total work. The total space required by the set of refinement trees is therefore $O(d^2 \log m)$. It is worth noting that the final suffix tree ST_D has a distinct leaf for each suffix of a pattern in $\mathcal{D}$. Hence Lemma 2 can be easily extended as follows:

Lemma 4. *Given a string $Y[1:y]$, Y's longest prefix occurring in a pattern of $\mathcal{D}$ (and its extended locus in ST_D) can be found in $O(\log y)$ time and $O(y)$ work.*

Step (2): Augment the suffix tree ST_D computing for each node $u \in ST_D$ the deepest ancestor of u, called $lp(u)$, which is the locus of a pattern in $\mathcal{D}$ (i.e., $W(lp(u)) \in \mathcal{D}$). To do this, we mark the root of ST_D and all of its leaves that are locus of some $P_i\$_i$. If the leaf storing a string $P_i\$_i$ is connected to its parent by an arc whose first labeling character is $\$_i$, then we delete the mark from the leaf and mark its parent (i.e., this node is the exact locus of the pattern P_i). From the properties of suffix trees [16], it immediately follows that $lp(u)$ is u's deepest marked ancestor. Therefore, we can use Lemma 3 (with $|T| = O(d)$) and compute $lp(u)$ in $O(\log d)$ time and $O(d)$ work.

We further augment ST_D computing a set of pointers $ext(c, u)$, for all $c \in \Sigma$ and $u \in ST_D$, defined as follows:

Definition 5. For each node $u \in ST_D$ and for each character $c \in \Sigma$, we define $ext(c,u) = v$ if and only if v is the extended locus in ST_D of the longest prefix of $cW(u)$ occurring in D (possibly $cW(u)$ itself).

Notice that $ext(c,u)$ is different from the pointer defined in [8], because in that case a pointer is defined for a character c and a node u only if the string $cW(u)$ occurs in D and thus its extended locus is defined. In our case, instead, it may be $|W(ext(c,u))| < W(u)+1$, because $cW(u)$ might not occur in any pattern of $\mathcal{D}$. We remark also that the augmented ST_D still requires $O(d)$ space, because $|\Sigma| = O(1)$ (by the hypothesis), and thus we have a constant number of *ext*-pointers leaving from each node in ST_D. We prove the following result:

Lemma 6. *For each node $u \in ST_D$ and for each character $c \in \Sigma$, the pointer $ext(c,u)$ can be computed in $O(\log m)$ time and $O(d \log m)$ total work on the CRCW PRAM.*

Proof. In Step (1), all the substrings of the patterns in $\mathcal{D}$ have been consistently labeled, and the corresponding set of refinement trees has been built accordingly. Given a node $u \in ST_D$ and a character $c \in \Sigma$, let us consider the string $\alpha = cW(u)$, and its substrings of length 2^q, for $0 \leq q \leq \log|\alpha|$. The substrings $\alpha[i : i + 2^q - 1]$, with $i > 1$, are actually substrings of $W(u)$, and thus they have been labeled in Step (1). Conversely, the names of substrings $\alpha[1 : 2^q]$, for all $0 \leq q \leq \log|\alpha|$, are not directly available (because we do not know even if α occurs in a pattern of $\mathcal{D}$). They are computed inductively by observing that $\alpha[1 : 2^q] = \alpha[1 : 2^{q-1}]\,\alpha[2^{q-1} + 1 : 2^q]$, where the substring $\alpha[2^{q-1} + 1 : 2^q]$ is entirely contained in $W(u)$ and thus its name is already known. Hence, we can label all of α's prefixes having power-of-two length in $O(\log|\alpha|) = O(\log m)$ sequential time by using the BB matrices previously adopted to label $\mathcal{D}$'s patterns. Finally, using Lemma 4, we search for α in the set of refinement trees built on D, thus finding the extended locus $ext(c,u)$ of the longest prefix of α that occurs in some pattern of $\mathcal{D}$. □

Furthermore, ST_D is preprocessed in $O(\log d)$ time and $O(d)$ total work to support constant-time LCA queries [18]. This way, given two arbitrary leaves $\ell, \ell' \in ST_D$, the longest common prefix between the two suffixes $W(\ell)$ and $W(\ell')$ can be computed in $O(1)$ sequential time by means of $LCA(\ell, \ell')$. Therefore we have:

Theorem 7. *Preprocessing phase requires $O(\log d)$ time and $O(d \log m)$ work on the CRCW PRAM. The total required space is $O(d^2 \log m)$.*

4 Answering Query(T)

We describe an approach that answers Query(T) based upon the information computed in Section 3 and available in the augmented ST_D. We first introduce a problem which arises in answering Query(T) and whose solution is used as a key tool in our parallel algorithm.

4.1 Left Extension problem

Let X be a substring of a pattern in $\mathcal{D}$. Clearly, X is consistently labeled and $|X| \leq m$. Furthermore, let $c_h, \ldots, c_1$ be a sequence of characters drawn from Σ. For $1 \leq i \leq h$, we define lcp_i as the longest prefix of the string $c_i \ldots c_1 X$ that occurs in some pattern of $\mathcal{D}$ (i.e., it occurs in D). The following proposition is easily provable:

Proposition 8. *lcp_i is the longest prefix of $c_i lcp_{i-1}$ that occurs in some pattern of $\mathcal{D}$.*

Proposition 8 highlights that lcp_{i-1} contains the whole information that suffices for computing lcp_i. The next step consists of solving efficiently the *Left Extension* problem defined as follows: For all $i = 1, \ldots, h$, retrieve the extended locus u_i of lcp_i in ST_D (notice that u_i exists since lcp_i occurs in D, by definition). This problem was studied in [9]. We propose below a simpler solution based upon *ext*-pointers in ST_D.

Algorithm-LEP(X, $c_h \ldots c_1$)

Step 1: Let u_0 be the extended locus of X in ST_D. Retrieve u_0 by searching for X in the refinement trees built on D. Since X is consistently labeled, u_0 can be retrieved in $O(\log |X|) = O(\log m)$ sequential time (by Lemma 4).

Step 2: For $i = 1, \ldots, h$, set $u_i := ext(u_{i-1}, c_i)$.

Before showing the correctness of Algorithm-LEP, we state an intermediate result.

Lemma 9. *Let Z be a string and node z be its* extended locus *in ST_D. For each character $c \in \Sigma$, the node $ext(c, z)$ is the extended locus of the longest prefix of cZ occurring in some pattern of $\mathcal{D}$.*

Proof. Let v be the *extended locus* of the longest prefix of cZ occurring in some pattern of $\mathcal{D}$. Recall that *ext*-pointers are computed only for the substrings of D that have exact locus in ST_D. Therefore, if $Z = W(z)$, then the lemma clearly follows by Definition 5. Otherwise (i.e., Z is a proper prefix of $W(z)$, and $W(p(z))$ is a proper prefix of Z), we do not have directly the *ext*-pointer for Z. Nevertheless, we can show that $v = ext(c, z)$.

Since Z is a proper prefix of $W(z)$, we have that cZ is a proper prefix of $cW(z)$. Thus v is an ancestor of $ext(c, z)$ in ST_D (by suffix tree's structure). By contradiction, assume that v is a *proper* ancestor of $ext(c, z)$, that is, $v \neq ext(c, z)$. Thus, $W(v)$ is a *proper* prefix of $W(ext(c, z))$. Now, since v is a node in ST_D, it has at least two outgoing arcs that have, for example, as first labeling characters c' and c'', with $c' \neq c''$. Thus, $W(v)c'$ and $W(v)c''$ occur in D. Let $W(v) = c\beta$, for some $\beta \in \Sigma^*$, then we may conclude that $\beta c'$ and $\beta c''$ also occur in D. By suffix tree's properties, β must therefore have locus u_β in ST_D. Moreover, β is a proper prefix of $W(z)$ (recall that $W(v)$ is a proper prefix of $W(ext(c, z))$), so that u_β is a *proper* ancestor of z, and $ext(c, u_\beta) = v$ (exact locus). From the definition of v, the longest prefix of cZ occurring in some pattern of $\mathcal{D}$ must be $c\beta$ and, by the hypothesis, $|\beta| < |Z| < |W(z)|$. Hence, the longest prefix of $cW(z)$ occurring in some pattern of $\mathcal{D}$ should be $c\beta$, contradicting the hypothesis that $v \neq ext(c, z)$! □

The correctness of Algorithm-LEP is proved by induction. The basis holds since u_0 is the extended locus of X (by definition). Let us set $lcp_0 := X$. By the inductive hypothesis, u_{i-1} is the extended locus of lcp_{i-1}, thus lcp_{i-1} is a prefix of $W(u_{i-1})$. From Proposition 8, it immediately follows that lcp_i is the longest prefix of $c_i lcp_{i-1}$ occurring in some pattern of $\mathcal{D}$. Hence, applying Lemma 9 (with $c = c_i$, $Z = lcp_{i-1}$ and $z = u_{i-1}$), we derive that $ext(c_i, u_{i-1})$ is the extended locus of lcp_i.

Theorem 10. *Left Extension problem defined on a sequence of characters $c_h, \ldots, c_1 \in \Sigma$ and on a substring X of some pattern in $\mathcal{D}$, can be solved in $O(h)$ sequential time once the extended locus of X in ST_D is given.*

4.2 The algorithm

It consists of four steps, called Preprocessing, Sampling, Left Extension, and Retrieval. Let us describe first their main features and then proceed to their detailed discussion.

In the Preprocessing step, we label only the text substrings of length 2^q which start at positions $(h2^q + 1)$, for all $0 \leq q \leq \log m$ and $0 \leq h \leq \lfloor \frac{t}{2^q} \rfloor$. These substrings are $O(t)$ in

total. In the Sampling step, we consider a subset $\mathcal{S}$ of text positions which are $O(\log m)$ positions apart each other (note that $|\mathcal{S}| = O(\frac{t}{\log m})$). For each $i \in \mathcal{S}$, we compute the longest prefix $T[i : i + L_i - 1]$ of $T[i : t]$, for a proper value L_i, occurring in some pattern of $\mathcal{D}$ by using the refinement trees built on D and the text consistent labeling (see [9, 17] for a different approach). In the Left Extension step, we compute $T[j : j + L_j - 1]$ for all the other positions $j \in [1 : t]$, by exploiting the *ext*-pointers stored in the nodes of ST_D (see Theorem 10). In the Retrieval step, we use *lp*-pointers and substring $T[j : j + L_j - 1]$, for all $j = 1, \ldots t$, to retrieve the longest pattern in $\mathcal{D}$, say $P_{long(j)}$, that is prefix of $T[j : t]$.

PREPROCESSING STEP. Recall that m is the length of the longest pattern in $\mathcal{D}$. Let us assume to append to the end of T, t special symbols $\$ \notin \Sigma$, and $\$ \neq \$_j$, for $j = 1, \ldots, k$. This way $T = T[1 : 2t]$. We perform a "partial naming" of T by applying the labeling procedure of [6] to all text substrings $T[h2^q + 1 : h2^q + 2^q]$, where $0 \leq h \leq \lfloor \frac{t}{2^q} \rfloor$ and $0 \leq q \leq \log m$. For a fixed q, these substrings cover entirely T without any overlapping and thus their number is $O(\frac{t}{2^q})$. Hence, their total number is $O(t)$. Therefore, by using the BB matrices employed to label the patterns in $\mathcal{D}$, Preprocessing step takes $O(\log m)$ time and $O(t)$ work. We point out that, we are saving space by avoiding the labeling of those text substrings that do not occur in any pattern of $\mathcal{D}$ (i.e., whose entries in the BB matrices are not initialized). Moreover, we are saving time and work by performing only a partial naming.

SAMPLING STEP. Let us consider the subset $\mathcal{S}$ of text positions defined as follows: $\mathcal{S} = \{h2^{\log\log m} + 1 : h = 0, 1, 2, \ldots, \lfloor \frac{t}{2^{\log\log m}} \rfloor\}$. That is, $\mathcal{S}$ is the set of every other $2^{\log\log m}$ positions in T, so that $|\mathcal{S}| = O(t/\log m)$. For each suffix $T[i : t]$, with $i \in \mathcal{S}$, the longest prefix $T[i : i + L_i - 1]$, for a proper value $L_i \geq 0$, occurring as a substring of some pattern in $\mathcal{D}$ satisfies the following property:

Proposition 11. $|P_{long(i)}| \leq L_i \leq m$.

Notice that, even if $T[i : t]$ could have been not completely labeled in the Preprocessing step, each substring $T[h2^q + 1 : h2^q + 2^q]$ that occurs in $T[i : i + L_i - 1]$ has been consistently labeled because $T[i : i + L_i - 1]$ is a substring of some pattern in $\mathcal{D}$ (by its definition) and $i = h'2^{\log\log m} + 1$, for some $h' \geq 0$. Hence, we can find $T[i : i + L_i - 1]$, searching for $T[i : t]$ in the refinement trees built on D (Lemma 2). Sampling step requires $O(\log m)$ time and $O(t)$ total work, since we are searching for $O(t/\log m)$ suffixes in total. We remark that, this step determines also the exact length L_i, for all $i \in \mathcal{S}$.

EXTENSION STEP. We compute the longest prefix $T[j : j + L_j - 1]$ occurring in some pattern of $\mathcal{D}$, for all the other positions $j \in [1 : t]$. Indeed, we exploit the *ext*-pointers stored in each node of ST_D, the length L_i (for each $i \in \mathcal{S}$), and the result proved in Theorem 10.

We map one processor to each position $i \in \mathcal{S}$ (hence we need $O(\frac{t}{\log m})$ processors in total). Each processor executes the following algorithm:

- Let u_i be the extended locus of $T[i : i + L_i - 1]$, where $i \in \mathcal{S}$ (determined in the Sampling step).
- **For** $s := 1$ **to** $2^{\log\log m} - 1$ **do**
  - $u_{i-s} := ext(T[i - s], u_{i-s+1})$.
  - Compute the length L_{i-s} of the longest prefix of $T[i - s : t]$ occurring in some pattern of $\mathcal{D}$, by using L_{i-s+1} and an LCA query. That is, if u_{i-s} is the root of ST_D then set $L_{i-s} := 0$; otherwise proceed as follows:
    - Let ℓ_{i-s+1} and ℓ_{i-s} be any two leaves of ST_D descending from u_{i-s+1} and u_{i-s}, respectively, and assume that $W(\ell_{i-s}) = P[r : |P|]$, for some pattern $P \in \mathcal{D}$.
    - Determine the new leaf ℓ associated with the second suffix of $P[r : |P|]$, i.e., $W(\ell) = P[r + 1 : |P|]$.
    - Set $L_{i-s} := 1 + \min\{L_{i-s+1}, |W(LCA(\ell_{i-s+1}, \ell))|\}$.

From Proposition 8, we have $L_{i-s+1} \geq L_{i-s} - 1$ and thus $T[i-s : i-s+L_{i-s}-1]$ is a prefix of the string $T[i-s : i-s+L_{i-s+1}]$. The computation of the node u_{i-s} is correct, as immediately derives from Theorem 10 because node u_{i-s} is the extended locus (maybe exact locus) of $T[i-s : i-s+L_{i-s}-1]$ (with $lcp_0 = T[i : i+L_i-1]$ and $lcp_s = T[i-s : i-s+L_{i-s}-1]$).

It remains to be shown that L_{i-s} is correctly computed by the LCA query executed in the above algorithm. Indeed, let us assume that u_{i-s} is not the root of ST_D, thus $L_{i-s} > 0$. Let $W(\ell_{i-s+1})$ and $W(\ell_{i-s})$ be two suffixes of some patterns in $\mathcal{D}$ associated with the two leaves ℓ_{i-s+1} and ℓ_{i-s}, respectively. From suffix tree's properties, we clearly have that $T[i-s+1 : i-s+L_{i-s+1}] = W(\ell_{i-s+1})[1 : L_{i-s+1}]$ and $T[i-s : i-s+L_{i-s}-1] = W(\ell_{i-s})[1 : L_{i-s}]$. From the observations above and since $L_{i-s+1} \geq L_{i-s} - 1 \geq 0$, we can state the following result:

Proposition 12. *$W(\ell)[1 : L_{i-s}-1]$ is a (maybe proper) prefix of $W(\ell_{i-s+1})[1 : L_{i-s+1}]$, where ℓ is the leaf storing the second suffix of $W(\ell_{i-s})$.*

Hence, two cases may arise in the computation of L_{i-s}, either $L_{i-s} = L_{i-s+1} + 1$, or $L_{i-s} < L_{i-s+1} + 1$. From Proposition 12, it is clear that both two cases are managed correctly by $LCA(\ell_{i-s+1}, \ell)$. For the time complexity of the Left Extension step, we observe that all of the $O(\frac{t}{\log m})$ processors can execute simultaneously the loop in $O(\log m)$ time (by Theorem 10). After that, for each suffix $T[j : t]$ we have the extended locus u_j of its longest prefix $T[j : j+L_j-1]$ occurring in some pattern of $\mathcal{D}$ and its length L_j.

RETRIEVAL STEP. We retrieve the longest pattern, say $P_{long(j)}$, that is prefix of $T[j : t]$. If u_j is the exact locus of $T[j : j+L_j-1]$ (i.e., $|W(u_j)| = L_j$) and u_j is the locus of some pattern in $\mathcal{D}$ (i.e., $W(u_j) \in \mathcal{D}$), then we set $P_{long(j)} := W(u_j)$. Otherwise, we set $P_{long(j)} := W(lp(u_j))$. The correctness derives from Proposition 11 and from the definitions of L_j and u_j. As far as for the time complexity, this step requires $O(1)$ sequential time for each position in T. Hence, using $O(t/\log m)$ processors, this step takes $O(\log m)$ time in total.

Summing up the time and work bounds required by the four steps above, we immediately derive:

Theorem 13. *Given an arbitrary text $T[1 : t]$, Query(T) can be answered in $O(\log m)$ time and $O(t)$ total work on the CRCW PRAM.*

Acknowledgments I thank R. Grossi and the anonymous referees for their helpful comments and suggestions on the early version of this paper.

References

1. A. V. Aho, and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 333–340, 1975.
2. A. Amir and M. Farach. Adaptive dictionary matching. In *IEEE Symposium on Foundations of Computer Science*, 760–766, 1991.
3. A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Science*, 208–222, 1994.
4. A. Amir, M. Farach, R. M. Idury, H. La Poutré, and A. A. Schäffer. Improved dictionary matching. *Information and Computation*, 258–282, 1995.
5. A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. In *Combinatorial Pattern Matching*, 259–272, 1992.
6. A. Apostolico, C. Iliopolus, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 347–365, 1988.

7. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 762–772, 1977.
8. M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In *Combinatorial Algorithms on Words*, 97–107, 1985.
9. M. Farach and S. Muthukrishnan. Optimal parallel dictionary matching and compression. In *ACM Symposium on Parallel Algorithms and Architectures*, 244–253, 1995.
10. P. Ferragina. Incremental Text Editing: a new data structure. In *European Symposium on Algorithms*, LNCS 855, 495–507, 1994.
11. P. Ferragina and F. Luccio. On the parallel Dictionary Matching problem: new results with applications. In *European Symposium on Algorithms*, 1996 (to appear).
12. R. Hariharan. Optimal parallel suffix tree construction. In *ACM Symposium on Theory of Computing*, 290-299, 1994.
13. J. Já Já. *An introduction to parallel algorithms.* Addison-Wesley, 1992.
14. R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. In *ACM Symposium on Theory of Computing*, 125–136, 1972.
15. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 63–78, 1977.
16. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 262–272, 1976.
17. S. Muthukrishnan and K. Palem. Highly efficient dictionary matching in parallel. In *ACM Symposium on Parallel Algorithms and Architectures*, 69–78, 1993.
18. B. Schieber and U. Vishkin. On finding lowest common ancestor: simplification and parallelization. *SIAM Journal on Computing*, 1253–1262, 1988.
19. P. Weiner. Linear pattern matching algorithm. In *IEEE Switch. Aut. Theory*, 1–11, 1973.