1991

Efficient CRCW-PRAM Algorithms Combining Multiple Autonomous Databases

Alberto Apostolico

Report Number:
91-009

EFFICIENT CRCW-PRAM ALGORITHMS COMBINING MULTIPLE AUTONOMOUS DATABASES

Alberto Apostolico

CSD-TR-91-009
February 1991
(Revised July 1991)

Efficient CRCW-PRAM Algorithms for Universal Substring Searching *

Alberto Apostolico†

July 9, 1991

Fibonacci Report 91.2

Abstract

A standard representation for strings is proposed, which has the following properties.

(1) For any string x, putting x in such a standard representation requires $O(\log|x|)$ CRCW-PRAM steps and $O(|x|\log|x|)$ total work and space.

(2) Let W be a collection of strings individually given in such a standard representation. Let w be an arbitrarily chosen string in W, w' an arbitrary substring of w, and $\{\bar{w}_1, \bar{w}_2, ..., \bar{w}_t\}$ an arbitrary set of substrings of strings in W. Then, a CRCW PRAM with $O(\bar{n} = \sum_{h=1}^{t} |\bar{w}_h| + |w'|)$ processors will find all the occurrences of w' in $\{\bar{w}_1, \bar{w}_2, ..., \bar{w}_t\}$, in constant time.

Key Words: Combinatorial Algorithms on Words, String Matching, Universal (Sub)string Searching, Parallel Computation, CRCW PRAM, Lexicographic Order, Squares and Repetitions in a String.

AMS subject classification: 68C25

---

*This research was supported, through the Leonardo Fibonacci Institute, by the Istituto Trentino di Cultura, Trento Italy.

†Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA and Dipartimento di Matematica Pura e Applicata, University of L' Aquila, Italy, axa@cs.purdue.edu. Additional support was provided in part by the National Research Council of Italy, 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.

1 Introduction

A number of sequential and parallel algorithms on strings [5] have been set up to date. The current version of a bibliography compiled by I. Simon [20] lists over 350 titles. In his recent survey of *string searching* algorithms [1], A. Aho references 140 papers. Algorithms on strings acquire information primarily by pairwise comparison of input characters. Typically, the number of character comparisons performed in the worst case is also the leading factor in the time and space complexity achieved by these algorithms. Most algorithms on strings do not assume or exploit any order relation on the input characters. In fact, alphabet order is quite often not even implied by the statement of the problem. The corresponding algorithms only expect to derive a result in $[=, \neq]$ from the comparison of any two symbols or strings. A handful of algorithms on strings, however, need results in $[<, =, >]$ from each alphabetic or lexicographic comparisons, in order to function. This includes of course all problems defined in terms of *lexicographic* orders. Among the problems in this class, we find that of sorting a set of strings over some ordered alphabet (e.g., in bucket sorting [2]), finding the lexicographically least circular shift of a string (e.g., in checking polygon similarity [21]), computing the Lyndon factorization of a string (e.g., in some public key cryptosystems [11]), etc.

While the assumption that the alphabet be ordered does not pose a restriction in practice (every practical encoding of an alphabet is subject to a natural order relation), it does represent sometimes a discriminating feature in string algorithmics. For example, some lower-bounds for string editing and related problems (see, e.g., [23]) have been established in a model of computation where only tests of equality between symbols are allowed. Outside this model, such lower-bound constructions no longer hold, even though no significant exploitation of the alphabet order is known.

Recently, a few algorithms have been produced which derive their increased efficiency precisely from the assumption that the alphabet be ordered [3, 10, 19]. This is quite interesting, since it shows that assuming an arbitrary order on the input alphabet may lead to discover a more efficient solution to problems on strings to which any notion of alphabet order seems totally extraneous. In this paper, we discuss some such algorithms in connection with the efficient parallel implementation of the following extension of the classical problem of string searching.

Assume we are given a set of strings W upon which we want to perform many *string queries*, as follows. In each query, we specify arbitrarily a substring w' of some string w in W (possibly, $w' = w$) as the *pattern*, ad also a set $W' = \{\bar{w}_1, \bar{w}_2, ..., \bar{w}_t\}$ of *textstrings*, where each $\bar{w}$ is a string from W or a substring of one such string. The result of the query is the set of *all* the occurrences of w' in W'.

This problem is solved by serial computation in time linear in the total length $\bar{n} = \sum_{h=1}^{t} |\bar{w}_h| + |w'|$ of the arguments of the query, by resorting to any of the available fast string searching algorithms (see, e.g., [14]). Alternatively, one may precompute an index such as the suffix tree [18] for every string w in W, at an individual cost of $O(|w| \log |w|)$, and then process each query w' in time proportional to $O(|w'| + q)$, where q is the number of occurrences of w' in the smallest subset of W from which the strings of W' were selected.

On a CRCW-PRAM, the algorithm [7] solves the problem in time $O(\log\log |w'|)$ with $(\bar{n}/\log\log |w'|)$ processors. A by-product of the suffix tree construction in [6] allows one to test whether or not a pattern y occurs (or to detect the first occurrence of y) in a text x in time $\log |y|$ with $|y|/\log |y|$ processors, but the preprocessing of x requires auxiliary space proportional to $|x|^2 \log |x|$. The algorithm in [22] performs our string query in $O(\log^* |w'|)$ time with $\bar{n}/\log^* |w'|$ processors, but it requires an $O(\log^2 |w'|/\log\log |w'|)$-time preprocessing of the pattern. Bringing this preprocessing in line with the time complexity of the processing phase is precluded by the log log-time lower bound recently established in [8] for parallel string searching.

In this paper, we develop a standard representation for strings supporting string queries in constant time. Specifically, we show that if all strings in W are given in their individual standard representations, then a CRCW PRAM with $O(\bar{n} = \sum_{h=1}^{t} |\bar{w}_h| + |w'|)$ processors can find all the occurrences of any w' in any set $W' = \{\bar{w}_1, \bar{w}_2, ..., \bar{w}_t\}$ from W, in constant time. Putting a string x in such a standard representation requires $O(\log |x|)$ CRCW-PRAM steps and $O(|x| \log |x|)$ total work and space. Thus, in particular, searching for *any substring* of a pattern of size m in *any substring* of a text of size n can be done in constant time with at most $n + m$ processors, once both the text and the pattern have been put in standard form at a cost of $O((n+m) \log n)$ operations. This has the same global complexity as the early algorithm in [13] which, however, only handled one definite pattern at a time.

In view of the recent results in [22] and [8], it may be instructive to interpret the present constructions in terms of tradeoff between precomputation and run-time computation. One consequence of such constructions, for instance, is that it is possible to preprocess a text x in $O(\log |x|)$ time with $|x|$ processors, preprocess a pattern y in constant (alternatively, $O(\log |y|)$) time with $|y|^2$ (alternatively, $|y|$) processors, and then answer any *substring* query on the pattern in constant time.

Single-pattern parallel searches do not translate into efficient *universal* searches of the kind considered here, in that they depend crucially on the specific pattern being considered. Tipically, these algorithms build a series of partitions of the text into larger and larger blocks keeping track for each block of which substrings beginning in that block match a longer and longer prefix of the pattern. At each stage, the processors assigned to each block check whether a suitable extension of each candidate substrings in the block still matches a prefix of the pattern. In order to translate any such strategy to the case of g different patterns, we would need g times as many processors as those used in that strategy, since earlier searches do not yield any advantage for the subsequent ones. Our approach consists instead of pre-computing some "universal" substrings in the texstring, which are used to latch on possible occurrences of any individual pattern. These substrings are universal in the sense that they reflect structural properties of the text, independently of any particular pattern. Reversing the usual perspective of string searching, we search for occurrences of such text substrings into any given pattern, and a careful organization of the latter represents the way in which the total work is reduced.

This paper is organized as follows. In the next section, we outline the basic criterion used in our constructions. For this, we adopt as a paradigm a special case of the standard, single-pattern search where the assumption is made that both the pattern and the text do not contain any substring in the form ww. In Sections 3 and 4, we generalize our single search to unrestricted cases, and present the standard representation of strings supporting instantaneous substring-searches after preprocessing.

We use the model of computation known as CRCW PRAM. The reader is referred to [3] for more details about this model, as well as for the conventions that support the fast partitions and allocations of processors presupposed in this paper. It should be noted that we make frequent use of the constant-time *priority write* emulation of the model described in [12].

2 Outlining the Main Criterion

We work with *words* or *strings* from an alphabet A ordered according to the linear relation $<$. This order induces a lexicographic order on A^+, which we also denote by $<$. Given two words u and v, we write $u \ll v$ or $v \gg u$ to denote that there are two symbols a and a' with $a < a'$, and a word $z \in A^*$ such that za is a prefix of u and za' is a prefix of v. Thus, $u < v$ iff either $u \ll v$ or u is a prefix of v. If $x = vwy$, then the integer $1 + |v|$, where $|v|$ is the length of v is the *(starting) position* in x of the *substring* w of x. Let $I = [i, j]$ be an interval of *positions* of a string x. We say that a substring w of x *begins* in I if I contains the starting position of w, and that it *ends* in I if I contains the position of the last symbol of w.

A string w is *primitive* if it is not a power of another string (i.e., writing $w = v^k$ implies $k = 1$). A primitive string w is a *period* of another string z if $z = w^c w'$ for some integer $c > 0$ and w' a possibly empty prefix of w. A string z is *periodic* if z has a period w such that $|w| \leq |z|/2$. It is a well known fact of combinatorics on words that a string can be periodic in only one period [17]. We refer to the shortest period of a string as *the* period of that string. A string w is a *square* if it can be put in the form vv in terms of a primitive string v (v is the *root* of the square). A string is *square-free* if none of its substrings is a square.

The basic idea subtending our algorithms is best explained in terms of the standard, single-pattern string searching problem. Let then y s.t. $|y| \geq 4$ be this pattern and x a text string, as in Fig. 1. Consider the ordered set $\mathcal{S}$ of all *positioned* substrings of y having length $c = 2^{(\lfloor \log |y| \rfloor - 2)}$, and let (i, s) be the one such substring such as s is a lexicographic minimum in $\mathcal{S}$ and i the smallest starting position of s in y. Substring (i, s) is called the *seed* of y. Pattern y is *left-seeded* if $i < c$, *right-seeded* if $i > |y| - 2c + 1$, *balanced* in all other cases. Let now the positions of x be also partitioned into *cells* of equal *size* $c = 2^{(\lfloor \log |y| \rfloor - 2)}$, and assume that there is at least one occurrence of y in x, starting in some cell B. In principle, every position of B is equally qualified as a candidate starting position for an occurrence of y. However, the same is not true for the implied occurrences of the seed of y. This seed will start in a cell B' that is either B itself or a close neighbor of B. Consider the set of all substrings of x which start in B' and have length $|s|$. It is not difficult to see then that the one such positioned substring corresponding to (i, s) has the property of being a lexicographic minimum

among all such substrings originating in B' and to its right, or originating in B' and to its left, or both, depending on whether y is left-, right-seeded, or balanced. Once we have a candidate position for s in B', it is trivial to check in constant time with s processors whether this actually represents an occurrence of y, since $|y| \leq 8|s|$. The problem is thus to identify such a candidate position. Note that, although we know that the seed of, say, a left-seeded pattern must be lexicographically least with respect to all substrings of equal length that begin in B' and to its right, there might be up to $|s| = |B'|$ substrings with this property. Even if we were given the starting positions of all such substrings (hereafter, *left stubs* [3]), checking all of them simultaneously might require $|s|^2$ processors.

However, assume for a moment that x and y were known to be square-free. Then, any pair of consecutive left stubs (i', z') and (i'', z'') in B' must differ on one of their first $i'' - i'$ symbols. Along these lines, it is possible to build, from the ordered sequence of left stubs, a corresponding ordered sequence of prefixes of left stubs with the following properties: (1) these prefixes are in strictly (i.e., according to the relation $\ll$) increasing lexicographic order from left to right, and (2) the sum of their lengths is bounded by $|B'|$ up to a small multiplicative constant. Point 2 is a handle to check all these prefixes against (i, s) simultaneously and instantaneously, with $O(|s|)$ processors. Point 1 guarantees that these prefixes are all different, whence at most one of the comparisons might return with equality. If this case occurs, we would have identified the unique candidate position j for a seed of y beginning in B'. As already mentioned, checking whether there is an actual occurrence of y seeded at j is done trivially in constant time with $O(|s|)$ processors.

In order to compute the sites of candidate seeds, we need the lists of starting positions of the left stubs in each block of x. These lists are called *left lists*, and they can be computed with $|x|$ processors in less than $\lfloor \log |y| \rfloor$ main *passes*, starting with the left lists trivially associated with the partition of positions of x into blocks of size 1. In each subsequent pass, the cells of the partition of the positions of x double in size, and the left lists relative to the blocks in the current partition are produced from a suitable composition of the old left lists in a pair of adjacent old blocks. The crucial task is to perform each pass in constant time with n processors. This rests on the following result from [3].

Theorem 1 *Let (B_d, B_{d+1}) be two consecutive cells in a partition of x. Given the left lists of B_d and B_{d+1}, the left list of $B_d \cup B_{d+1}$ can be produced*

by a CRCW PRAM with $|B_d \cup B_{d+1}|$ processors in constant time.

In conclusion, we can find all occurrences of a left-seeded, square-free pattern y in a square-free text x in $\log|y|$ time with $n = |x|$ processors. In the following sections, we generalize this construction to the case where x and y are not necessarily square-free. Throughout the rest of the paper, we concentrate on the management of left-seeded patterns, but it shall be apparent that the case of right-seeded patterns is handled by symmetric arguments.

3 A Standard Representation for Constant-time String Searching

In this section and in the following one, we describe a $O(\log|y|)$-time, $O(|x|\log|y|)$-work CRCW algorithm for detecting all the occurrences of a single pattern y in a text x. This performance matches that of the early parallel algorithm in [13], but is clearly inferior to the (optimal) $O(\log\log n)$ performance of the algorithm in, e.g., [7]). However, the new algorithm is actually more powerful than that of [13], in the sense that it yields, as an intermediate product, a standard representation for strings having the important property exposed in the following Theorem.

Theorem 2 *Consider a set W of strings individually given in standard representation. Let w be an arbitrarily chosen such string, w' an arbitrary substring of w, and $\{\bar{w}_1, \bar{w}_2, ..., \bar{w}_t\}$ an arbitrary set of substrings of strings in W. A CRCW PRAM with $\bar{n} = \sum_{h=1}^{t} |\bar{w}_h| + |w'|$ processors can find all the occurrences of w' in $\{\bar{w}_1, \bar{w}_2, ..., \bar{w}_t\}$ in constant time.*

The proof of Theorem 2 will be completed in Section 4. Theorem 2 is of interest in implementing parallel string data bases. Assume that every string in the data base is stored in its standard form. Whenever a new string w is added to the data base, we need to spend $O(\log|w|)$ CRCW time with $|w|$ processors in order to put w in standard form. Having done that once, however, we can answer in constant time any query regarding the occurrences of any substring w' of w in any set of substrings of any present or future strings (in standard form), including w itself, with a number of processors linear in the length of the arguments of the query.

To start with the discussion of Theorem 2, we need to recapture some notions and results from [3].

Let x be a string of n symbols, and $B = [h, h+m]$, where $m \leq n/2$ and $h \leq (n-2m+1)$, a cell of size m. Let $\mathcal{L}(B) = \{(i_1, z_1), (i_2, z_2), ..., (i_k, z_k)\}$ be the sequence of left stubs of B, and let the sequence $\{i_1, i_2, ..., i_k\}$ be the left list of B. (It will be clear that the cases where $k \leq e$ with e an arbitrary constant can be handled trivially in our strategy, thus we assume henceforth that $k > 3$.)

As an example, let the substring of x in block B be *eacaccdlacdacdacdlllhf* (cf. Fig. 2), and assume for simplicity that the positions of x falling within B be in [1,22]. We have 8 left stub in B, beginning with the rightmost such stub $(22, z_8) = f...$. Since h and l are both larger than f, the next left stub is $(17, z_7) = dlllhf...$. We immediately have $(16, z_6) = cdlllhf...$ and $(15, z_5) = acdlllhf...$. Since d and c are both larger than a, there will not be a left stub until $(12, z_4) = acdacdlllhf...$. We similarly have $(9, z_3) = acdacdacdlllhf...$. Finally, we have $(4, z_2) = accdlacdacdacdlllhf...$ and $(2, z_1) = acaccdlacdacdacdlllhf...$. Note that the prefix of z_1 of length $2 = i_2 - i_1$ matches the corresponding prefix of z_2. Similarly, the prefix of z_3 of length $3 = i_4 - i_3$ matches a prefix of z_4, and the prefix of z_4 of length $3 = i_5 - i_4$ matches a prefix of z_5. We say that z_1 and z_2 are in a run, and so are z_3, z_4 and z_5.

Formally, for $f = 1, 2, ..., k-1$ in the left list, let l_f be the prefix of z_f such that $|l_f| = i_{f+1} - i_f$, and let the *mate* of (i_f, l_f) be (i_{f+1}, l'_f), where l'_f is the prefix of z_{f+1} having the same length as l_f. A substring $(i_j, z_j), (i_{j+1}, z_{j+1}), ..., (i_{j+f}, z_{j+f})$ with $f > 1$ of $\mathcal{L}(B)$ is a *run* if $l_j = l_{j+1} = ... = l_{j+f-1} = l'_{j+f-1} \neq l_{j+f}$. Stub (i_j, z_j) is called the *head* of the run. Run (i_j, z_j), (i_{j+1}, z_{j+1}), ..., (i_{j+f}, z_{j+f}) is *maximal* if $j = 1$ or $l_{j-1} \neq l_j$.

We now describe a sequence $\bar{\mathcal{H}}(B) = \{(h_1, \bar{r}_1), (h_2, \bar{r}_2), ..., (h_q, \bar{r}_q)\}$, where $\{h_1, h_2, ..., h_q\}$ is called the *head list* and is a subsequence of the left list $\{i_1, i_2, ..., i_k\}$, and $\bar{r}_f$ $(f = 1, 2, ..., q)$ is a suitable prefix of the left stub z_t at position $i_t = h_f$. Sequence $\bar{\mathcal{H}}(B)$ plays an important role in our constructions, because it is the shuffle of two sequences of words $\bar{\mathcal{H}}^{(1)}(B)$ and $\bar{\mathcal{H}}^{(2)}(B)$ with the following properties [3].

Lemma 1 *The word terms in* $\bar{\mathcal{H}}^{(p)}(B)$ $(p = 1, 2)$ *form a lexicographically strictly increasing sequence.*

Lemma 2 *The sum of the lengths of the word terms in* $\bar{\mathcal{H}}(B)$ *is bounded above by* $4|B|$.

The specification of the h's and $\bar{r}$'s requires some auxiliary notions, that are given next with the help of Fig. 2.

If $[(i_j, z_j), (i_{\hat{j}}, z_{\hat{j}})]$ is an ordered pair of (not necessarily consecutive) left stubs, we use w_j to denote the prefix of z_j of length $\hat{j} - j$. An ordered pair $[(i_j, z_j), (i_{\hat{j}}, z_{\hat{j}})]$ with the property that w_j is a prefix of $z_{\hat{j}}$ is called a *diplet.* A diplet is *strong* if $\hat{j} = j + 1$, i.e., if $w_j = l_j = l'_{j+1}$, *weak* otherwise. It is possible to prove that two left stubs are in a run if they form a diplet, whence a left stub cannot be the head of two distinct runs. (Note that this does not forbid that the last stub in a run be also the head of another run.)

We define now $\mathcal{H}(B) = \{(h_1, \bar{z}_1), (h_2, \bar{z}_2), ..., (h_q, \bar{z}_q)\}$ to be the ordered sequence of all left stubs that are not in any run and all run heads of maximal runs. In our previous example, we have 5 elements in $\mathcal{H}(B)$, starting at $h_1 = 1, h_2 = 9, h_3 = 16, h_4 = 17$ and $h_5 = 22$. Given a copy of x, the set $\mathcal{H}(B)$ is completely specified by the ordered sequence of starting positions of its elements, which we called the head list. The head list of any cell B enumerates also the starting positions of all elements of $\bar{\mathcal{H}}(B)$. Any such element can be identified by local manipulations, i.e., by comparing the respective values of three alternately consecutive elements of the head list. Observe that a maximal run (i_j, z_j), (i_{j+1}, z_{j+1}), ..., (i_{j+f}, z_{j+f}) is describable in compact form by giving i_j, its period length $|l_j| = i_{j+1} - i_j$, and and i_{j+f}. In the following, we consider the head list always annotated in this fashion (an empty annotation denoting a left stub not belonging to a run), so that the left list is implicitly described by it. This new version of head list is called *lexicographic* list.

For $f = 1, 2, ..., q-2$, let r_f be the prefix of $\bar{z}_f$ such that $|r_f| = h_{f+2} - h_f$, and let the *mate* of (h_f, r_f) be (h_{f+2}, r'_f), where r'_f is the prefix of $\bar{z}_{f+2}$ having the same length as r_f. Finally, define $\bar{\mathcal{H}}(B) = \{(h_1, \bar{r}_1), (h_2, \bar{r}_2), ..., (h_q, \bar{r}_q)\}$ as follows. First, set $\bar{r}_1 = r_1$, $\bar{r}_2 = r_2$, $\bar{r}_{q-1} = r'_{q-3}$ and $\bar{r}_q = r'_{q-2}$. Next, for $2 < f < q - 1$, set $\bar{r}_f = r'_{f-1}$ if $h_{f+2} - h_f < h_f - h_{f-2}$, and $\bar{r}_f = r_f$ otherwise. We now partition $\bar{\mathcal{H}}(B)$ into two subsequences $\bar{\mathcal{H}}^{(1)}(B)$ and

$\bar{\mathcal{H}}^{(2)}(B)$, each one of which is obtained by extracting alternate elements of $\mathcal{H}(B)$. Thus, $\bar{\mathcal{H}}^{(1)}(B) = \{(h_1, \bar{r}_1), (h_3, \bar{r}_3), (h_5, \bar{r}_5), ...\}$, and $\bar{\mathcal{H}}^{(2)}(B) = \{(h_2, \bar{r}_2), (h_4, \bar{r}_4), (h_6, \bar{r}_6), ...\}$ (cf. Fig. 2). In the following, we retain the indexing of the sequence $\bar{\mathcal{H}}(B)$ when speaking of either sequence $\bar{\mathcal{H}}^{(1)}(B)$ and $\bar{\mathcal{H}}^{(2)}(B)$. Thus, the subscripts of two consecutive elements in $\bar{\mathcal{H}}^{(1)}(B)$ differ by 2.

Let now w be a string, and assume w.l.o.g. that $|w|$ is a power of 2. For each $t = 1, 2, 4, ...$, let the positions of w be partitioned into $|w|/2^{t-1}$ disjoint cells each of size 2^{t-1}. Assume that the lexicographic lists relative to each cell partition are given. Clearly, the space required for storing all these lists is $O(|w| \log |w|)$. A string w together with the first ($\imath \leq \lfloor \log |w| \rfloor - 2$) lexicographic lists is said to be in *$\imath$-standard form.* When $\imath = \lfloor \log |w| \rfloor - 2$, we simply say that w is in *standard* form. We are now ready to show that searching for a string in standard form into another string also in standard form is done instantaneously with a linear number of processors. With y denoting the pattern and x the text, we revisit the discussion of the previous section.

Clearly, retrieving the seed (i, s) of y from its $|s|$-standard form is immediate. In fact, consider the partition of y into cells of size $|s|$ and let C be the cell of this partition which contains i.

Lemma 3 *Stub (i, s) is the first element of $\mathcal{H}(C)$.*

Proof: Straightforward, since word s is by definition the earliest lexicographic minimum among all substrings relative to the stubs in C. □

Lemma 3 is the handle to identify the position i of s in y. Since there are at most 4 cells in the partition of the positions of y, and each such cell contributes one known candidate, mutual comparison of the substrings of length $|s|$ starting at these candidate positions is all is needed. This is easily done in constant time with $|y|$ processors. Although there are more direct ways of computing the seed of y within these bounds, reasoning uniformly in terms of standard forms has other advantages in our context.

Assume now that there is an occurrence of a left-seeded pattern y starting in a cell B of the partition of x into cells of size $|s|$, and let B' be the cell of x where the corresponding occurrence of the seed s begins (cf. Fig. 1). We address the identification of the position j of s within B'. Clearly, j is

the position of a left stub in $\mathcal{L}(B')$. Lemma 1 of the previous section tells us that, if we consider the sequence, say, $\bar{\mathcal{H}}^{(1)}(B')$, then we can find in the general case either at most one term $\bar{r}_f$ such that $\bar{r}_f$ is a prefix of s, or two consecutive terms $\bar{r}_f$ and $\bar{r}_{f+2}$ such that $\bar{r}_f \ll s \ll \bar{r}_{f+2}$. The same would hold had we considered $\bar{\mathcal{H}}^{(2)}(B')$ instead. We thus search in, e.g., $\bar{\mathcal{H}}^{(1)}(B')$ for a pair of consecutive terms $\bar{r}_f$ and $\bar{r}_{f+2}$ such that, letting $\tilde{s}$ be the prefix of s of length $|\bar{r}_f|$, we have that $\bar{r}_f \leq \tilde{s} \ll \bar{r}_{f+2}$ (finding just $\bar{r}_f$ is sufficient if $\bar{r}_f$ is the last element of $\bar{\mathcal{H}}^{(1)}(B')$). Lemma 2 tells us that $O(|B|)$ processors are enough to match, simultaneously, each $\bar{r}_f$-term against a corresponding prefix $\tilde{s}$ of s. The technique in [12] allows us to obtain the lexicographic outcome of each such comparison in constant time.

Once $\bar{r}_f$ has been identified, the search for j is delimited to within the span of some run. Specifically, assume that $\bar{r}_f$ has a predecessor $\bar{r}_{f-1}$ and a successor $\bar{r}_{f+1}$ in $\bar{\mathcal{H}}(B')$. We need to search the runs headed by $\bar{z}_{f-1}$, $\bar{z}_f$ and $\bar{z}_{f+1}$ in order to identify the candidate position j of the seed y. In principle, we may have two or more identical stubs in a run, i.e., j may be anyone of several positions of left stubs that match s. Actually, more than one among such positions may correspond to an occurrence of the seed of y. A *full run* is defined as a run $(i_q, z_q), (i_{q+1}, z_{q+1}), ..., (i_{q+p}, z_{q+p})$ such that $z_h = z_{h+1}$ for $h = q, q+1, ..., q+p-1$. It is known [3] that there can be at most one full run in $\mathcal{L}(B)$.

Lemma 4 *Assume that $j - i + 1$ and $j' - i + 1 > j - i + 1$ are two distinct occurrences of y in x. Then, there is a full run in $\mathcal{L}(B')$ with $z = s$.*

Proof: By definition, s is a lexicographic minimum among the substrings of y of length $|s|$. Since y is left-seeded, then every member of $\mathcal{L}(B')$ that starts at a position equal to j' or higher is a substring of y. Therefore, (j', s) is a left stub. Now, stubs (j, s) and (j', s) have length $|s| = |B'|$, but their starting positions differ at most by $|B'| - 1$. Hence, these stubs form a diplet, and thus they belong to the same run. This run is a full run where $z = s$. □

Consider first the case where $\mathcal{L}(B')$ does not contain a full run. This implies that at most one candidate seed position j may exist in B'. Observe that any non-full run implies the existence of a substring of x of length not larger than $2|s|$ and in the form $u^k u' a$, where a is a character, u' is a prefix of u but $u'a$ is not. Knowing the head of this run and its immediate successor

in $\mathcal{L}(B')$, we know $|u|$, and we can compute k and $|u'a|$ by lexicographic comparison of strings of length $2|s|$. Similar computations are carried out on the suffix of y that starts at i. This leads to identify another integer k' and a string $u''a'$ such that u'' is a prefix of u but $u''a''$ is not, and $u^{k'}u''a'$ is a prefix of s'. Comparing now $u'a$ with $u''a'$ and k with k' identifies the unique position j of the left list possibly compatible with an occurrence of y. Assume that such a candidate position j was found. Since we know i, we can now compare $x_{j-i}x_{j-i+1}...x_{j-i+m}$ with y and output an occurrence of y at $j-i$ if the two strings match. These manipulations take constant time with $\Theta(B')$ processors, hence constant time with $\Theta(|x|)$ processors for the entire string x.

The case where we have a full run may yield more than one candidate only under the additional condition that the pattern is periodic with a period u equal to the period of this run. (Note that such a condition is trivially checked in constant time with $O(|y|)$ processors.) If the pattern does not have such a period, then the only candidate for an occurrence of the seed in this cell of x is the first stub in the full run. If the pattern does have such a period, then those occurrences of y that have seeds in B' are spaced apart precisely by $|u|$ positions, like the corresponding seed occurrences in B'. We can easily check all the occurrences relative to these seeds: specifically, we first perform substring comparisons similar to the above to discover the extent of a periodicity of the form u^k in a neighborhood of B' of size $|y|+|B'|$, and then we use this notion in order to compute all occurrences of y seeded in B' at once. Also these manipulations take constant time with $\Theta(B')$ processors, hence constant time with $\Theta(|x|)$ processors for the entire string x.

The case of a right-seeded pattern is handled somewhat symmetrically, i.e., by introducing lists of suitably defined *right stubs*, etc.

Let now y' be a substring of y, and consider the $(\log\lfloor|y'|\rfloor - 2)$-th lexicographic list for y. Clearly, y' is embedded in a set of at most 9 consecutive cells in the associated cell partition of y. The same holds for every occurrence of y' in any substring x' of x such that $|x'| \geq |y'|$. Assume to fix the ideas that y' is left-seeded. Note that if y' and its seed (i', s') start in the same cell, say, C' on y, it is no longer necessarily true that (i', s') is the first term in the head list of C'. However, (i', s') must still be a left stub in C'. Since the starting position f of y' in y is known, all we need to do is to identify the leftmost left stub in $\mathcal{L}(C')$ that starts at f or to the right of

f. This takes constant time with the priority-write emulation in [12], after which we have a suitable substitute for Lemma 3. From this point on, the search for y' into x' involves only minor variations with respect to the above description, and so does the search for y' in any set of substrings of a given set of strings. This concludes the discussion of Theorem 2. □

4 Epilogue

In this section, we reconsider briefly the task of searching from scratch for all occurrences of a string y into another string x. Along the lines of the preceding discussion, we may regard this task as subdivided in two phases. The task of the first phase (preprocessing) is to identify the seed of y and the positions of candidate, say, left-seeded patterns in x, the task of the second phase is to check these candidates. In principle, the preprocessings of y is less demanding than that of x, but in view of Theorem 2 it is advantageous to treat the two strings uniformly and speak in terms of a generic string w. The goal of the preprocessing is to put w in standard form. This is done by a simple doubling scheme that consists of approximately $\lfloor \log |w| \rfloor$ stages.

At the beginning of stage t ($t = 1, 2, ...$) of the preprocessing the positions of w are partitioned as earlier into $|w|/2^{t-1}$ disjoint cells each of size 2^{t-1}. Starting with the first cell $[1, 2^{t-1}]$, we give now all cells consecutive ordinal numbers. For $t = 1, 2, ...$, stage t handles simultaneously and independently every pair (B_{od}, B_{od+1}) of cells such that od is an odd index. The task of a stage is to build the lexicographic list relative to each cell $B_d \cup B_{d+1}$, using the lexicographic lists of B_{od} and B_{od+1}). This is supported by Theorem 1 and by the following additional result from [3].

Theorem 3 *Given the head lists of B_d and B_{d+1}, the head list of $B_d \cup B_{d+1}$ can be constructed in constant time by a CRCW PRAM with $|B_d|$ processors.*

In conclusion, we can list the following claim.

Theorem 4 *For any string w and integer $\ell \leq |w|$, a CRCW PRAM with $|w|$ processors can compute the lexicographic lists relative to the first $\log \ell$ stages of the preprocessing of w in $O(\log \ell)$ time and using linear auxiliary space per stage.*

The constructions that lead to theorems 1 and 3 are too elaborate to be reported here. Still, some insight can be gained in the light of the discussion of the preceding section. For example, it is not difficult to see that the head list of B_{d+1} is usually a suffix of $B_d \cup B_{d+1}$. Thus, the combined list can be produced by first finding the prefix of $\mathcal{L}(B_d)$ where the first element of $\mathcal{L}(B_{d+1})$ "falls" lexicographically, and then appending $\mathcal{L}(B_{d+1})$ to such prefix. In analogy to the search stage, the first step can be implemented by simultaneously checking which terms of $\bar{\mathcal{H}}^{(p)}(B_d)$ $(p = 1, 2)$ matches a prefix of the first left stub of B_{d+1}. Lemma 1 tells us that at most two of those terms might return with a match. Lemma 2 tells us that $\Theta(m)$ processors are sufficient to carry out this task in constant time.

To conclude this section, we list two additional applications of theorems 2 and 4.

In on-line string searching, we are interested in performing efficiently queries involving many different patterns on a same textstring. In this context, it may be advantageous to preprocess the text once and for all if this speeds up the subsequent queries significantly. We can use standard forms as a space-efficient alternative in the on-line string searching described in [6]. The preprocessing of that method has the same time complexity but requires $\Theta(|x|^2 \log |x|)$ auxiliary space. The present preprocessing requires $\Theta(|x| \log |x|)$ such space. The on-line pattern-processing phase in [6] requires $\log |y|$ steps with $|y|/\log |y|$ processors, after which it outputs *whether or not* y occurs in x. Given the text in standard form, the standardization of the pattern with the present approach requires the same time with $O(|y|)$ processors. After that, we can find *all the occurrences* of y in any substring x' of x in constant time with $O(|x'|)$ processors.

Additional possible uses of Theorem 2 arise in the context of approximate string searching in parallel, as described, for instance, in [15] and [6], where we are interested in answering repeatedly and quickly questions of the kind: given a suffix of the text and a suffix of the pattern, what is the longest prefix that these two suffixes have in common?

Acknowledgement

I am indebted to C. M. Hoffmann for some enlightening remarks.

References

1. Aho, A.V. (1990), "Algorithms for finding Patterns in Strings" in *Handbook of Theoretical Computer Science*, to appear.

2. A. V. Aho, J. E. Hopcroft, J. D. Ullman (1974), "*The Design and Analysis of Computer Algorithms*", Addison-Wesley.

3. Apostolico, A. (1989), "Optimal Parallel Detection of Squares in Strings", *Algorithmica*, to appear.

4. Apostolico, A. and M. Crochemore (1989), "Optimal Canonization of All Substrings of a String", *Information and Computation*, to appear.

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

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

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

8. Breslauer, D. and Z. Galil (1990), A Lower Bound for Parallel String Matching, *Proc. 23rd ACM Symp. on Theory of Computation*, to appear.

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

10. Crochemore, M. and D. Perrin (1989), "Two-way Pattern Matching", TR 89-2, Université Paris Nord, *J. ACM*, to appear.

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

12. Fich, F.E., R. L. Ragde and A. Wigderson (1984), "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.

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

14. Knuth, D.E., J.H. Morris and V.R. Pratt (1977), "Fast Pattern Matching in Strings", *SIAM Journal on Computing* **6**, 2, 323-350.

15. Landau, G.M and U. Vishkin (1989), "Fast Parallel and Serial Approximate String Matching" *Journal of Algorithms* **10**, 2, 157-169.

16. Lothaire, M (1983), *Combinatorics on Words*, Addison-Wesley, Reading, Mass.

17. Lyndon, R.C. and M.P. Shutzenberger (1962), "The Equation $a^M = b^N c^P$ in a Free Group", *Michigan Mathematical Journal* **9**, 289-298.

18. McCreight, E.M. (1976), "A Space-economical Suffix Tree Construction Algorithm", *Journal of the ACM* **23**, 262-272.

19. Manber, U. and G. Myers (1990), "Suffix Arrays: A New Method for On-Line String Searches", *Proceedings of the 1-st SODA*, 319-327.

20. Simon, I. (1990), "Palavras, Automatos e Algoritmos - Uma Bibliografia" University of Sao Paulo.

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

22. Vishkin, U. (1990), Deterministic Sampling - A New Technique for Fast Pattern Matching, *SIAM J. Computing* **20**, 1, 22-40.

23. Wong, C.K. and A.K. Chandra (1976), "Bounds for the String Editing Problem", *Journal of the ACM* **23**, 1, 13-16.

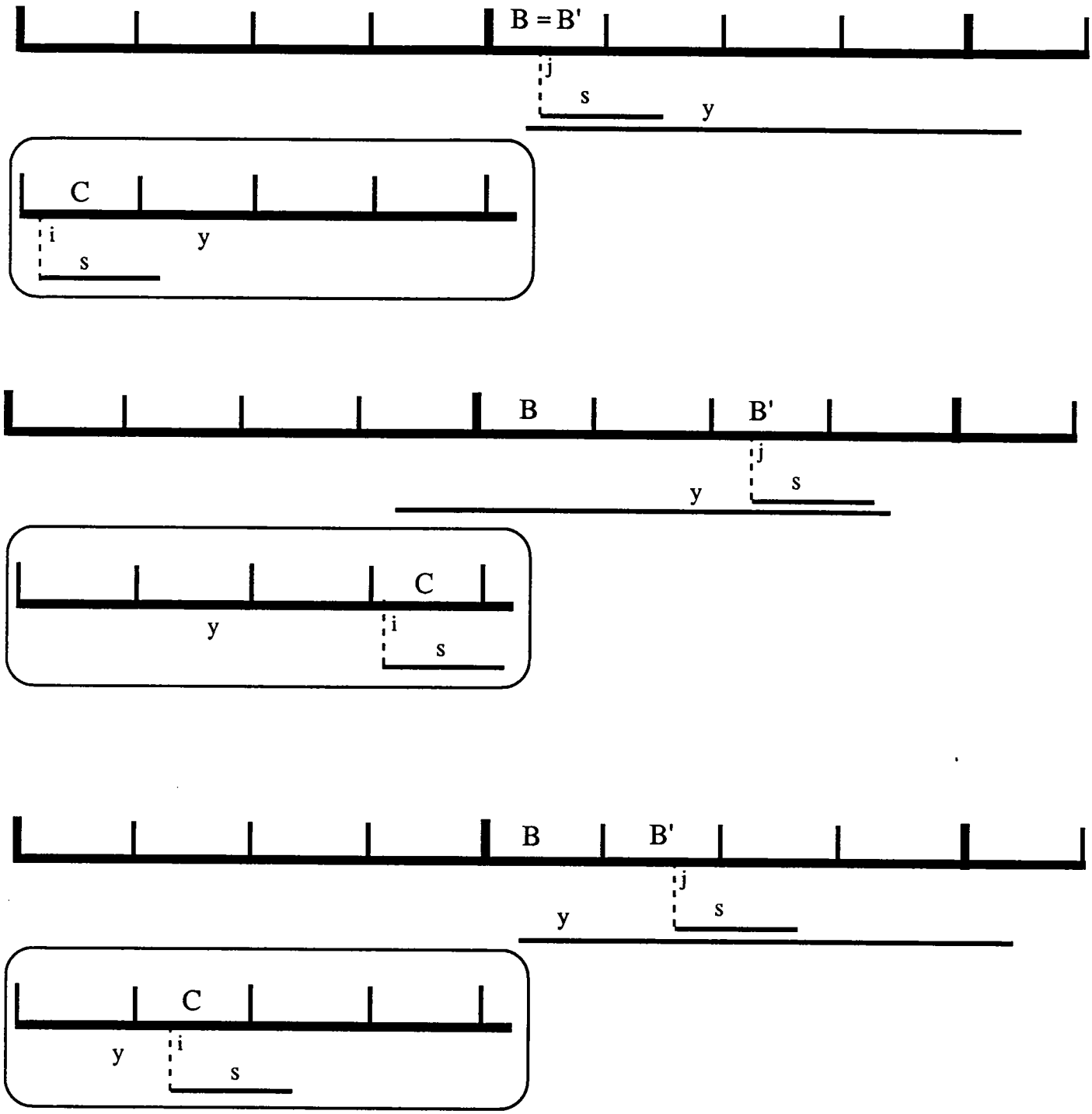

Figure 1
Illustrating the main criterion as applied to a left-seeded (top portion of the figure), right-seeded (middle) and balanced pattern (bottom).

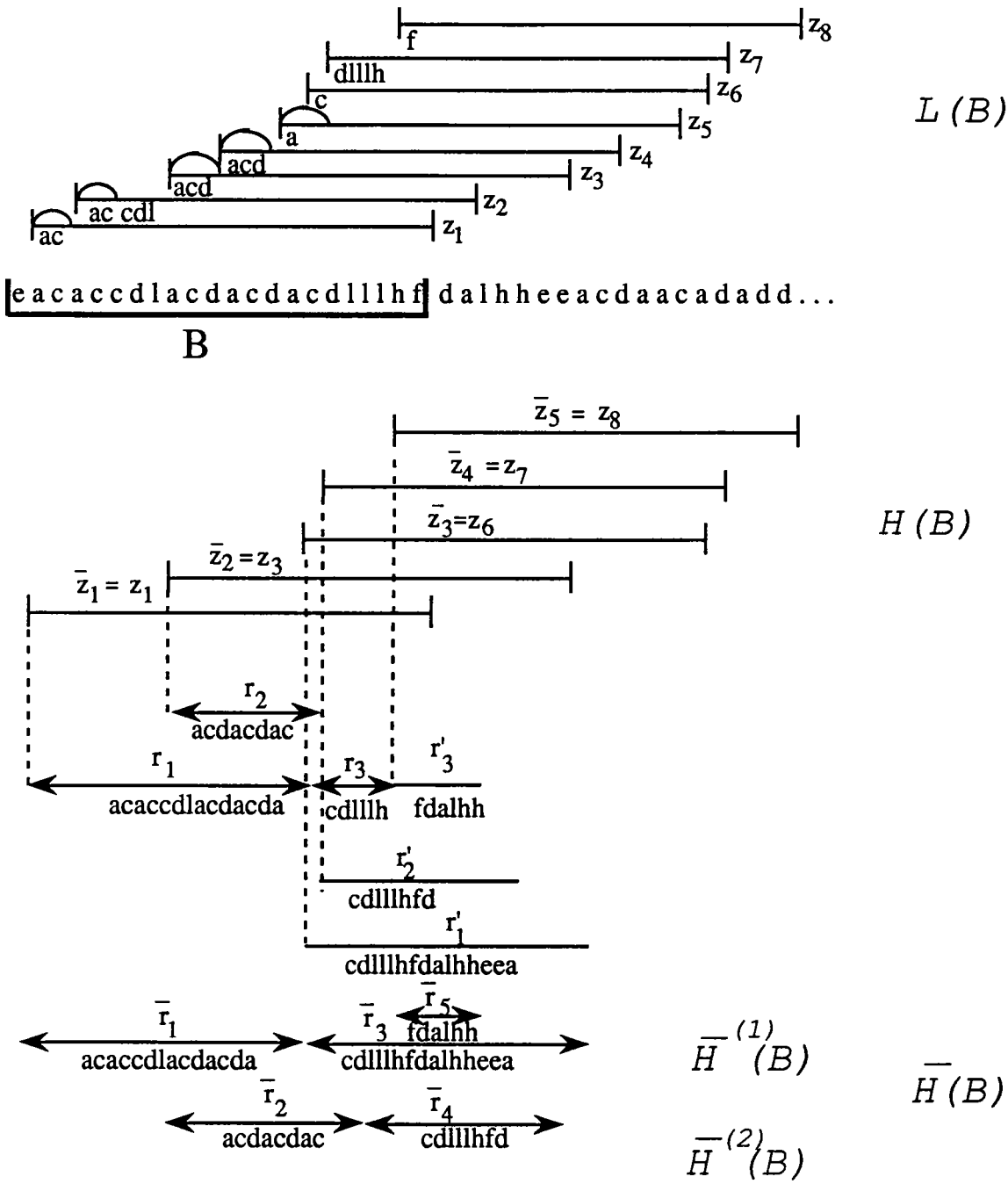

Figure 2
Extracting the sequences H (top half of the figure) and $\bar{H}$ (bottom half) from a sequence of left stubs.