On a parallel-algorithms method for string matching problems (overview)

Suleyman Cenk Sahinalp * *Uzi Vishkin* †

Abstract

Suffix trees are the main data-structure in string matching algorithmics. There are several serial algorithms for suffix tree construction which run in linear time, but the number of operations in the only parallel algorithm available, due to Apostolico, Iliopoulos, Landau, Schieber and Vishkin, is proportional to $n \log n$. The algorithm is based on labeling substrings, similar to a classical serial algorithm, with the same operations bound, by Karp, Miller and Rosenberg. We show how to break symmetries that occur in the process of assigning labels using the Deterministic Coin Tossing (DCT) technique, and thereby reduce the number of labeled substrings to linear.

*Department of Computer Science, University of Maryland at College Park

†University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742; and Dept. of Computer Science, Tel Aviv University, Tel Aviv, Israel; Partially supported by NSF grants CCR-8906949 and CCR-9111348.

1 Introduction

Suffix trees are apparently the single most important data-structure in the area of string matching.

We present a parallel method for constructing the suffix tree T of a string $S = s_1 \dots s_n$ of n symbols, with s_n being a special symbol \$ that appears nowhere else in S. We use A to denote the *alphabet* of S. The suffix tree T associated with S is a rooted tree with n leaves such that:

(1) Each path from the root to a leaf of T represents a different suffix of S.

(2) Each edge of T represents a nonempty substring of S.

(3) Each nonleaf node of T, except the root, must have at least two children.

(4) The substrings represented by two sibling edges must begin with different characters.

An example of a suffix tree is given in Figure 1.

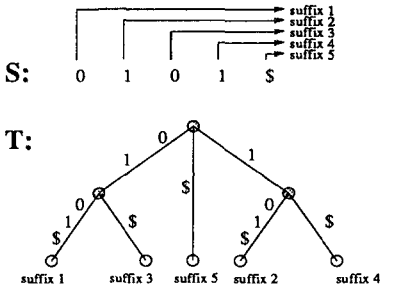

Figure 1: Suffix tree T of string $S = 0\ 1\ 0\ 1\ \$$

Serial algorithms for suffix tree construction were given in [KMR72], [We73], and [Mc76]. The two latter algorithms achieve a linear running time for an alphabet whose size is constant.

A parallel algorithm was given in [AILSV88].

A Symmetry Breaking Challenge: As in the algorithm of [KMR72] work complexity of the above mentioned parallel algorithm is $O(n \log n)$. The approach of [KMR72] and [AILSV88] does not lend itself to linear work for the following reason: As these algorithms progress, they label **all** $n-1$ substrings of size 2, then all $n-3$ substrings of size 4, and in general all $n - 2^i + 1$ substrings of size 2^i $(1 \leq i \leq \log n)$. This results in a number of labels which is proportional to $n \log n$ and this dictates the work complexity. The extra logarithmic factor in the label-count is due to the increasing redundance among these substrings (because of the overlaps), as they become longer. The problem is that there has been no consistent way for selecting only one among a subset of overlapping substrings, since they all "look-alike". The main new idea of this paper is in introducing a solution to this *symmetry breaking*

problem. Our **most interesting concrete result** is in being able to build a suffix tree using only a linear number of labels.

The general area of string matching has been enriched by parallel methods that enabled new serial algorithms as in this paper. Previous examples include [Ga85], [Vi85], and [Vi91]. The new method is also relevant for sequence analysis in compressed data, since it allows for consistent compression of data. This can be done in the context of parallel or serial algorithms. Applications of the new method for data compression will be discussed in the full version.

The method described in this paper leads to several incomparable complexity results as described in [SV93]. We quote here only one which can be derived with reasonable effort from our description. For an alphabet whose size is polynomial in n, the method gives an $O(n \log^* n)$ work algorithms and $O(n^\epsilon)$ time for any constant $0 < \epsilon \leq 1$.

2 The Algorithm

2.1 High-level Description

The algorithm works in two stages. In the **first stage** we attach **labels** to various substrings of S, recognizing some identities. This is done in iterations. In iteration 1, S is partitioned into blocks of size 2 or 3 characters. Each block is labeled with a number between 1 and n, in a way which satisfies the following two **consistency properties**:

Partition-consistency (we state this property informally) Let X_i be a "long enough" substring of S (starting) at location i and let X_j be a substring at location j, which is equal to X_i; then, with the exception of some margins, X_i and X_j will be partitioned in the same way.

Label-consistency All blocks consisting of the same string of characters will get the same label.

An example of consistent partitioning and consistent labeling is given in Figure 2.

So, iteration 1 "shrinks" $S = S(0)$, into a new string $S(1)$, reducing its length by a factor of at least two. Subsequent iterations apply the same procedure. Iteration i, $i = 2, 3, \ldots$ shrinks string $S(i-1)$ into string $S(i)$ satisfying similar partition-consistency and label-consistency properties. The size of strings $S(i)$ will be at most $n/2^i$.

The **second stage** is devoted to constructing the suffix tree of S in iterations. The input for the last iteration is $T(1)$, which is the suffix tree of (some kind of) labels which are derived from $S(1)$. The last iteration constructs the suffix tree of $S(0) = S$. The i'th-prior-to-the-last iteration constructs the suffix tree $T(i)$ (the suffix tree of labels derived from $S(i)$), by using $T(i+1)$.

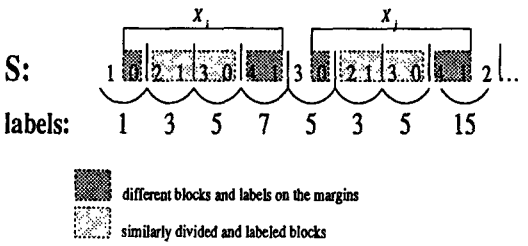

Figure 2: Consistent partitioning, margins, and consistent labeling

2.2 First Stage

Let R, a string of characters (of size m), be the input for an iteration of the **first stage**. The iteration partitions R into blocks of size 2 or 3 and labels each block. The problem is how to do it to satisfy partition-consistency (defined formally later) and label-consistency. We first describe the main steps of an iteration, and then give a detailed description.

The main steps of an iteration:

Each character r_i checks if it is in a substring of length 2, or more, of a single repeated character. If yes (i.e, $r_i = r_{i-1}$ or $r_i = r_{i+1}$), it uses procedure LENGTH-BASED partitioning (or, procedure LENGTH-BASED for short) for obtaining the block partitioning. If no, it uses procedure CONTENT-BASED partitioning (or, procedure CONTENT-BASED for short) for obtaining the block partitioning. (*Comment*: The case were some character r_i is not part of a substring of a single repeated character but its previous character r_{i-1} as well its next character r_{i+1} are both part of such substrings, need a separate treatment which is suppressed in this presentation.)

The reader is asked to be alert to the following: in order to better convey the main ideas of our algorithm, we **systematically suppressed, or deferred** dealing with the important case of substrings consisting of a single repeated character. Dealing with this case is relatively simpler, and it does not change the essentials of most of the new techniques presented in this paper.

We start with describing procedure CONTENT-BASED. Consider a substring $R_D = r_\alpha \dots r_\beta$, where $r_i \neq r_{i+1}$, for $\alpha \leq i < \beta$. Namely, we do not allow a substring of the form aa. The main idea behind the *partitioning procedure* below is the use of the *deterministic coin tossing technique* of Cole and Vishkin [CV86a] for dividing R_D into blocks.
(*Comment.* Our use of the deterministic coin tossing technique is novel. We will use it for deriving "signatures" of strings, mapping similar substrings to the same signature. The only previous paper which made use of this technique for producing signatures is apparently by Mehlhorn, Sundar and Uhrig [MSU94]. They limited these signatures to compare full strings, and did not have to deal either with consistency relative to substrings, or to consider assigning signatures for strings of repeated characters (i.e. in the form aaa...).)

1. Put a divider to the left of r_α and to the right of r_β. Each instruction for putting a divider below should be augmented with the following *caveat*: we actually put the new divider only if it does not create a block consisting of a single character.

2. Put a divider to the right of $r_{\alpha+1}$. Put a divider to the left of $r_{\beta-1}$.

3. For each character r_i of R_D compute tag_i, the index of the least significant bit of r_i which is different than r_{i+1} in binary representation. If r_{i+1} does not exist (for now, this can happen only if $i = m$), set $tag_i := 0$.

4. For each character r_i of R_D, compare tag_i with tag_{i+1} and tag_{i-1}. If any of them does not exist (for now this can happen if $i = 1$, or $i = m$), take the non-existing value to be 0. For all "strict local maxima" (i.e., $tag_i > tag_{i+1}$ and $tag_i > tag_{i-1}$) put a block divider between r_i and r_{i+1}. For all "weakly local maxima" (i.e. $tag_i \geq tag_{i+1}$, and $tag_i \geq tag_{i-1}$), put a block divider between r_i and r_{i+1}, if bit $diff_i$ of r_i is 1.

5. For each substring of R_D, which lies between two dividers, do the following. If the substring has ≤ 3 elements those elements quit. Otherwise, for each character r_i, replace character r_i by tag_i, and recursively apply the CONTENT-BASED procedure separately to each substring (which lies between two dividers).

Example Let $R = \ldots 3\ 3\ 8\ 4\ 2\ 1\ 2\ 4\ 8\ 4\ 8\ 4\ 8\ 8 \ldots$. Typically, we will apply the CONTENT-BASED procedure to a longest substring which satisfies the input conditions which in this case will be:
R_D = 8 4 2 1 2 4 8 4 8 4 . After applying steps 1 and 2 we have:
|8 4|2 1 2 4 8 4|8 4| . In binary representation R_D becomes:
1000 0100|0010 0001 0010 0100 1000 0100|1000 0100 , and the corresponding *tag* values are:
3 2 1 1 2 3 3 3 3 0 . Hence in the first round of CONTENT-BASED, we partition R_D as follows:
8 4|2 1 2 4|8 4|8 4 . Then we apply CONTENT-BASED procedure again to get:
8 4|2 1|2 4|8 4|8 4 , as the final partitioning of R_D.

Lemma 1.1 (the Block-partitioning Lemma): The above partitioning procedure divides R_D into blocks of size 2 or 3.

Proof: The Lemma follows from known facts about the *deterministic coin tossing technique* .

Lemma 1.2 (the Consistency Lemma): Let R_D be a substring of R in which no two successive characters are identical, and let R'_D be another substring of R which is identical with R_D. The CONTENT-BASED procedure partitions R_D and R'_D so that all but (at most) $\log^* n + 1$ blocks in the right margin and (at most) $\log^* n + 1$ blocks in the left margin are identical.

Proof: The lemma follows by a simple induction on the recursive call of CONTENT-BASED on R_D and R'_D. Designating a character as a local minima in call number p (which determines the location of a block divider) depends on at most $2p+3$ neighboring characters to its left and $2p + 3$ neighboring characters to its right and we only have $\log^* n$ successive recursive calls for CONTENT-BASED.

Next, we proceed to consider a substring of a single repeated character $R_R = (r_{j+1} =$

$\ldots = r_{j+k}$) which cannot be further extended. That is, $r_j \neq r_{j+1} = r_{j+2} \ldots = r_{j+k} \neq r_{j+k+1}$.

We describe procedure LENGTH-BASED for obtaining a block-partitioning of R_R in Appendix 1. By applying these procedures one after another, we obtain a complete partitioning of R.

Now, we can attach labels to blocks preserving label-consistency, and finish the iteration.

Each block is labeled with a number between 1 and m to satisfy label-consistency. This can be done in a CRCW PRAM in $O(1)$ time with $O(m)$ work in two substeps. In the first substep we attach labels to all blocks of size 2, as well as to the first two characters of all blocks of size 3. This is done using an $m \times m$ array L. For each such two-character substring ij, where $1 \leq i, j \leq m$, the location of i is entered into entry $L(i, j)$. This is similar to [AILSV88]. In the second substep we attach labels to all block of size 3. For each three-character substring ijk, we obtain its label by applying the first substep to fk where f denotes the label of ij as computed in the first substep. The space complexity is $O(m^2)$. This can be reduced to $O(m^{1+\epsilon})$, for any fixed $\epsilon > 0$, as in [AILSV88]. Getting linear space using hashing (and thereby entering randomization), as suggested in [MV91], is also a possibility.

An example for block partitioning and labeling is given in Figure 3.

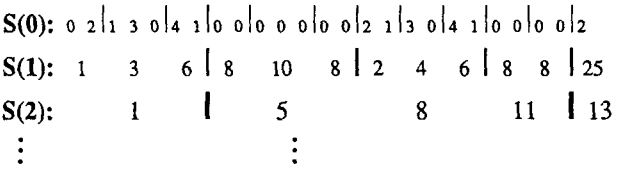

Figure 3: Block division and labeling for successive iterations

2.3 Second Stage

Before starting to describe the **Second Stage**, we mention the issues which will be suppressed in this presentation: (1) Data structures for representing the suffix tree and keeping the labels of substrings. (2) Processor allocation issues. (3) The case of substrings of a repeated character; so, for following our presentation the reader should assume that we never get two successive identical characters in any of the $S(k)$ sequences (for $k = 1, 2, \ldots,$).

Each iteration of the first stage provides a partition of the string into blocks and assigns labels to these blocks. the partitions in the first stage into k-substrings and the k-labels provided some limited similarities among substrings, as characterized by Lemma 1.2. For the construction of suffix trees later, it will be helpful to first develop another system of substrings of S (instead of the k-substrings) called **cores**, and label them with core **names**.

Any k-substring S_k induces another substring of the input which is called a k-**core**. Since

S_k is a substring of its k-core we say that S_k **spans** its k-core. An example of a core is given in Figure 4.

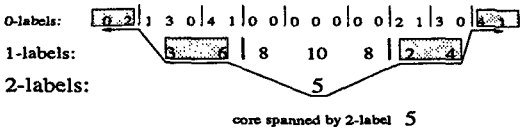

Figure 4: An example of a core for S in Fig.3; for illustration purposes, assume $2\log^* n + 3 = 2$ (which is not possible)

Given a substring S_k, we show how to **extend** it to the left and to the right to obtain a k-core. S_k is in the middle. To its left there will be $2\log^* n + 3$ strings which are $(k-1)$-substrings. To their left there will be will be again $2\log^* n + 3$ strings which are $(k-2)$-substrings, and so on till finally there will be $2\log^* n + 3$ which are (0)-substrings (they are, in fact, singletons). Finally, a k-core has also a symmetric "staircase" to the right of S_k. So, a k-core is a "double staircase" as illustrated in Figure 5.

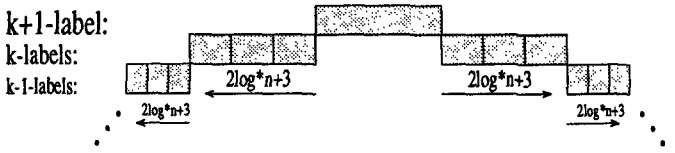

Figure 5: A k-substring is extended by $2\log^* n+3$, $k-1$-substrings followed by $2\log^* n+3$, $k-2$-substrings... towards both left and right to obtain a k-core

The suffix (of the original pattern) which begins at the leftmost character of a k-core is called the **suffix of that core**, and is referred to as a **k-suffix.** (*Comment*: Each $(k+1)$ suffix is also a k-suffix. In other words, the suffix of every $(k+1)$-core is suffix of a k-core, since the leftmost character of the $(k+1)$-core is also the leftmost character of the k-core.

STEP 2.1 Corresponding to each iteration $k = 1, 2, \ldots$ of the **first stage**, do the following. For each k-substring, compute the k-core it spans and label each such core with a core name, which is called a **k-name.** Each k-core is actually a concatenation (with overlaps) of $4\log^* n + 7$, or $4\log^* n + 6$, $(k-1)$-cores; and the name of the k-core is derived from the names of these $(k-1)$-cores. A key point in deriving the k-names is that this is done **consistently**; that is, if the $k-1$-names form an identical sequence elsewhere then both k-cores will get the same k-name.

STEP 2.2 The suffix tree is constructed iteratively. In iteration k of the **Second Stage**, the suffix tree $T(k)$ **of k-cores using k-names** is built. This suffix tree has limited resolution, as some possible identical prefixes between two $(k+1)$-cores cannot be precisely expressed with $(k+1)$-names. **Iteration k builds $T(k)$ from $T(k+1)$ by improving the resolution with a more dense set of cores which are shorter.** Note that the

iterations are numbered backwards, where the final iteration (which is iteration 1) gives the desired suffix tree.

The main ideas in the Second Stage We need to do several things in order to build $T(k)$ from $T(k+1)$.

1. **Get tree** $T(k)_0$. This is still the suffix tree of the $(k+1)$-suffixes (as $T(k+1)$), but it uses k-names. Procedure REFINE will derive tree $T(k)_0$ from tree $T(k+1)$.

Procedure REFINE will work as follows. Some $(k+1)$-cores represented in $T(k+1)$ are replaced with a concatenation (with overlaps) of the $4\log^* n + 9$, or $4\log^* n + 8$, k-cores that form them. In parallel for every edge of $T(k+1)$ advance step-by-step through its first $4\log^* n + 8$ (or $4\log^* n + 7$) k-cores, merging identical names into a single edge.

Observation: The common prefixes of k-cores between any two sibling edges in $T(k+1)$ can not be longer than $4\log^* n + 8$ (or $4\log^* n + 7$) as the definition of the $k+1$-core implies; hence the procedure REFINE will discover all similarities between $k+1$-cores in terms of k-cores in at most $4\log^* n + 8$ steps.

2. **Get tree** $T(k)_1$. For each $(k+1)$-core do the following. The $(k+1)$-core is spanned by some $(k+1)$-substring S_{k+1}. To the left of S_{k+1}, there will be $2\log^* n + 3$ strings which are k-substrings. Consider the next k-substring. An example is given in Figure 6. Pick the

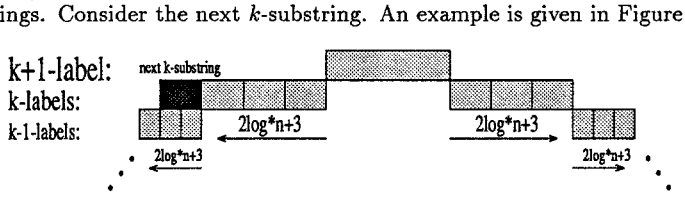

Figure 6: An illustration of the next k-substring of a $k+1$-core

k-core that it spans. Given all the k-cores that were picked, we build the suffix tree of their suffixes using k-names and denote it $T(k)_1$. This is done as follows. Divide the k-cores into equivalence classes, and let us focus on one of these equivalence classes, to be denoted Q. In $T(k)_1$, the suffixes of the cores in Q will be represented, as follows. From the root, there will be an outgoing edge labeled with the common k-name of Q, which will lead to a node. The subtree rooted at this node will have a leaf for each suffix having the core-name of Q as its prefix. It can be readily observed that each of these suffixes corresponds to some suffix (and a leaf) of $T(k)_1$. We will apply procedure CONTRACT below to $T(k)_0$ to include only these leaves and only those internal nodes which split into two (or more) paths in $T(k)_0$ leading from the root to two (or more) of these leaves.

Procedure $CONTRACT(T, s_1, s_2, \ldots, s_m)$
Input: A rooted tree $T(k)_0$. For each internal node of the tree, its children, whose number is denoted by n_{k+1}, are given in an array of size α. Assume that each node v has its distance from the root, denoted $level(v)$. We also assume that tree $T(k)_0$ has been preprocessed so that a query requesting the *lowest common ancestor* of any pair of nodes in $T(k)_0$ can be processed in constant time ([BV88], [SV88]). The input also includes $s_1, s_2, \ldots, s_m$, a subset of the leaves of T, given in the same order as they appear in T.

Output: a contracted version of $T(k)_0$ with the leaves $s_1, s_2, \ldots, s_m$.

3. **Get tree** $T(k)_2$. Consider the k-substrings whose k-cores were picked for $T(k)_1$ above. Consider the next k-substring. Pick the k-core that it spans, unless it is a suffix of some $(k+1)$-core. Given all the k-cores that were picked, we build the suffix tree of their suffixes using k-names and denote it $T(k)_2$. This is done using $T(k)_1$ similar to the construction of $T(k)_1$ using $T(k)_0$.

4. Use procedure MERGE to merge the three trees $T(k)_0, T(k)_1$, and $T(k)_2$ into $T(k)$, which is the suffix tree of the suffixes of all k-cores, using k-names. Procedure MERGE works in $4\log^* n + 10$ rounds. Starting simultaneously from the roots of $T(k)_0, T(k)_1$, and $T(k)_2$ it advances step-by-step through the first $4\log^* n + 10$ k-names in each of them, merging identical names into a single edge. By the first $4\log^* n + 10$ k-names in each tree, we mean exploring in each tree all the paths starting from the root and in each path consider the first $4\log^* n + 10$ k-names.

Perhaps the **most interesting observation in this paper** is that this gives the desired $T(k)$. This is implied by the following.

Observation. Take two k-suffixes that come from two different trees among $T(k)_0, T(k)_1$, and $T(k)_2$. Then, they can share a prefix of at most $4\log^* n + 10$ k-cores.

Proof (for $k \geq 1$ only). This is implied by Lemma 1.2 as follows. Assume that the common prefix of two k-suffixes that come from two trees among $T(k)_0, T(k)_1$, and $T(k)_2$, is longer than $4\log^* n + 10$ k-cores. Take the first such $4\log^* n + 10$ k-cores, and focus on the chain of the $4\log^* n + 10$ k-substrings that span them. They are identical in the two k-suffixes. By Lemma 1.2, the $(k+1)$-substrings that cover (well over $\log^* n$ of the) k-substrings in the middle section of this chain are also identical. Finally, consider the leftmost full $(k+1)$-cores in each of the k-suffixes. It is not hard to see that the $(k+1)$-suffix of each of these $(k+1)$-cores hits the two k-suffixes at the the same distance (which is 0, 1, or 2) from their leftmost k-core. Therefore, both k-suffixes belong to the same tree among $T(k)_0, T(k)_1$, and $T(k)_2$. The proof for $k = 0$ is similar; actually, it becomes an important ingredient in proving the correctness of our whole algorithm.

References

[AILSV88] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, Parallel Construction of a Suffix Tree with Applications, In *Algorithmica*, 3: 347–365, 1988.

[BV88] O. Berkman, and U. Vishkin, Recursive star-tree parallel data-structure, In *SIAM J. Computing*, 22,2: 221–242, 1993.

[BDHPRS89] P. C. P. Bhatt, K. Diks, T. Hagerup, V. C. Prasad, T. Radzik, and S. Saxena, Improved Deterministic Parallel Integer Sorting, In *Information and Computation*, 94: 29–47, 1991.

[BLMPSZ91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J Smith, M. Zagha, A Comparison of Sorting Algorithms for the Connection Machine CM-2, In *Proceedings of the* 3^{rd} *Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, 1991.

[CV86a] R. Cole, and U. Vishkin, Deterministic Coin Tossing with Applications to Parallel List Ranking, In *Information and Control*, 70: 32–53, 1986.

[CV86b] R. Cole, and U. Vishkin, Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms, In *Proceedings of the* 18^{th} *Annual ACM Symposium on the Theory of Computing*, pages 206–219, 1986.

[Ga85] Z. Galil, Optimal Parallel Algorithms for String Matching, In *Information and Control*, 67: 144–157, 1985.

[KMR72] R. M. Karp, R. E. Miller, and A. L. Rosenberg, Rapid Identification of Repeated Patterns in Strings, Trees, and Arrays, In *Proceedings of the* 4^{th} *Annual ACM Symposium on the Theory of Computing*, pages 125–136, 1972.

[MV91] Y. Matias, and U. Vishkin, On Parallel Hashing and Integer Sorting, In *Journal of Algorithms*, 12,4: 573–606, 1991.

[Mc76] E. M. McCreight, A Space - Economical Suffix Tree Construction Algorithm, In *Journal of the ACM*, 23: 262–272, 1976.

[MSU94] K. Mehlhorn, R. Sundar, and C. Uhrig, Maintaining Dynamic Sequences under Equality - Tests in Polylogarithmic Time, to appear In *Proceedings of the* 5^{th} *Annual ACM - SIAM Symposium on Discrete Algorithms*, 1994.

[RR89] S. Rajasekaran, and J. H. Reif, Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms, In *SIAM Journal of Computing*, 18: 594–607, 1989.

[SV93] S. C. Sahinalp, and U. Vishkin, Symmetry Breaking in Suffix Tree Construction, In preparation

[SV88] B. Schieber, and U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization, In *SIAM Journal of Computing*, 17: 1253-1262, 1988.

[Vi85] U. Vishkin, Optimal Parallel Pattern Matching in Strings, In *Information and Control*, 67: 91–113, 1985.

[Vi91] U. Vishkin, Deterministic Sampling - A New Technique for Fast Pattern Matching, In *SIAM Journal of Computing*, 20: 22–40, 1991.

[We73] P. Weiner, Linear Pattern Matching Algorithm, In *Proceedings of the* 14^{th} *IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Appendix 1: Procedure LENGTH-BASED

We tentatively put a block divider to the (immediate) right of r_{j+k} and to the left of r_{j+1}, thereby separating the block partition of $r_{j+1} \ldots r_{j+k}$ from its surroundings. (Later, we may remove the tentative dividers put above if we find out that there is a divider to the left of r_j or to the right of r_{j+k+1}.)

We break into several cases depending on the remainder of k modula 4.
Case 1: k is an even number. Put block dividers to the right of each of $r_{j+2}, r_{j+4}, \ldots, r_{j+k}$.
Case 2: $k-1$ is divisible by 4. The middle element of R_R is $r_{j+(k+1)/2}$. Put block dividers to the right of $r_{j+2}, r_{j+4}, \ldots, r_{j+(k-1)/2}$ and $r_{j+(k+5)/2}, r_{j+(k+9)/2}, \ldots, r_{j+k-2}$; so that the middle element is in a block of size 3 and all other blocks are of size 2.
Case 3: k+1 is divisible by 4. Put block dividers to the right of $r_{j+2}, r_{j+4}, \ldots, r_{j+(k-3)/2}$ and $r_{j+(k+3)/2}, r_{j+(k+7)/2}, \ldots, r_{j+k-2}$; so that again the middle element is in a block of size 3 and all other blocks are of size 2.

Procedure LENGTH-BASED actually continues to process R_R in iterations till it reduces to a single label. In each of the subsequent iterations the reduced length of R_R at the time will be treated according to the case analysis above; this is done in spite of the fact that a middle label may be different than other labels.

Example Let $R = \ldots 10000000002 \ldots$, the procedure LENGTH-BASED starts from the beginning and ending location of substring of 0s and partitions R accordingly ($Case2$ applies):
$\ldots 1|00|00|000|00|2 \ldots$

This ends procedure LENGTH-BASED.