BRICS RS-95-11 Apostolico & Breslauer: Detecting Squares in a String

BRICS

Basic Research in Computer Science

An Optimal $O(\log\log n)$ Time Parallel Algorithm for Detecting all Squares in a String

Alberto Apostolico
Dany Breslauer

BRICS Report Series **RS-95-11**

ISSN 0909-0878 **February 1995**

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

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

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

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

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

An Optimal $O(\log \log n)$ Time Parallel Algorithm for Detecting all Squares in a String*

Alberto Apostolico[†] Dany Breslauer[‡]

Abstract

An optimal $O(\log \log n)$ time concurrent-read concurrent-write parallel algorithm for detecting all squares in a string is presented. A tight lower bound shows that over general alphabets this is the fastest possible optimal algorithm. When p processors are available the bounds become $\Theta(\lceil \frac{n \log n}{p} \rceil + \log \log_{\lceil 1+p/n \rceil} 2p)$. The algorithm uses an optimal parallel string-matching algorithm together with periodicity properties to locate the squares within the input string.

1 Introduction

A nonempty string of the form xx is called a *repetition*. Some strings, such as $a^n = aaa \cdots aa$, contain $\Omega(n^2)$ repetitions since they have $\Omega(n)$ repetitions starting at most positions. A *square* is defined as a repetition xx where x is primitive[1]. Strings that do not contain any repetition are called *repetition-free* or *square-free*. For example, '*aa*', '*abab*' and '*baba*' are squares which are contained in the string '*baababa*'.

It is trivial to show that any string whose length is larger than three over alphabets of two symbols contains a square. However, there exist strings of infinite length on three letter alphabets that are square-free, as shown by Thue [29, 30] at the beginning of the century. Since then, numerous works have been

*The results presented in this paper were were first reported at the 19th International Colloquium on Automata, Languages and Programming [3].

†Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA and Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy. Partially supported by NSF Grants CCR-89-00305 and CCR-92-01078, by NATO Grant CRG 900293, by the National Research Council of Italy, and by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM).

‡BRICS – Basic Research in Computer Science, Centre of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark. Partially supported by ESPRIT Basic Research Action Program of the EC under contract #7141 (ALCOM II). Parts of the research reported in this paper were carried out while the author was supported by the IBM Graduate Fellowship while studying at Columbia University, New York, NY, while visiting at the Istituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, with the support of the European Research Consortium for Informatics and Mathematics postdoctoral fellowship, and while visiting at the Università de L'Aquila, L'Aquila, Italy.

[1]A string x is primitive if $x = u^k$ for some integer k implies that $k = 1$ and $x = u$.

published on the subject and repetitions in strings have been found relevant to several fields, including coding theory, formal language theory, data compression and combinatorics [1, 6, 7, 14, 15, 22, 23, 28].

The alphabet that the input symbols are chosen from has an important role in the design of efficient string algorithms. The literature distinguishes between four types of alphabets: *constant size alphabets* that have a bounded number of symbols; *fixed alphabets* where the symbols are assumed to be integers from a restricted range; *ordered alphabets* where the alphabet is (arbitrarily) totally ordered and the only access an algorithm has to the input symbols is by order comparisons; and *general alphabets* where the only access an algorithm has to the input symbols is by equality comparisons.

In the last decade, several sequential algorithms that find all squares in strings have been published. Algorithms that were discovered by Apostolico and Preparata [4] and by Crochemore [13, 15] find all squares in a string of length n over ordered alphabets in $O(n \log n)$ time. Rabin [27] gave a randomized algorithm that takes $O(n \log n)$ expected time over constant size alphabets. Any sequential algorithm that lists all squares in a string of length n must take at least $\Omega(n \log n)$ time, since there exist strings, such as the *Fibonacci strings* [13], that contain $\Omega(n \log n)$ distinct squares.

Main and Lorentz [25] discovered an algorithm that finds all squares in strings over general alphabets in $O(n \log n)$ time. They also proved that over general alphabets $\Omega(n \log n)$ comparisons are necessary even to decide if a string is square-free. In another paper, Main and Lorentz [26] show that the problem of deciding whether a string is square-free can be solved in $O(n)$ time over constant size alphabets. Crochemore [15] also gave a linear time algorithm for the latter problem.

In parallel, algorithms by Crochemore and Rytter [16, 17] test if strings over ordered alphabets are square-free in $O(\log n)$ time using n processors. These algorithms use $O(n^{1+\epsilon})$ space. Apostolico [2] designed an algorithm that tests if a string is square-free and also detects all squares within the same time and processor bounds using linear auxiliary space. Apostolico's algorithm [2] assumes that the alphabet is ordered, a restriction that is not necessary to solve this problem. Apostolico's algorithm for testing if a string is square-free is more efficient over constant size alphabets and achieves the $O(\log n)$ time bound using only $n/\log n$ processors. All these parallel algorithms are designed for the CRCW-PRAM computation model.

A parallel algorithm is said to be *optimal*, or to achieve an *optimal speedup*, if its time-processor product, which is the total number of operations performed, is equal to the running time of the fastest sequential algorithm for the same problem. All the parallel algorithms that are mentioned above achieve an optimal speedup. Notice that squares can be trivially detected in constant time using a polynomial number of processors; our goal is to develop parallel algorithms that are efficient with respect to both time and processor complexities.

In this paper we develop an optimal parallel algorithm that finds all squares in a string in $O(\log \log n)$ time. The new algorithm not only improves on the previous best bound of $O(\log n)$ time, but it is also the first efficient parallel algorithm for this problem over general alphabets. We derive a lower bound that shows that over general alphabets this is the fastest possible optimal algorithm by a reduction to a lower bound that was given by Breslauer and Galil [11] for the string matching problem. If p processors are available, then the bounds become $\Theta(\lceil \frac{n \log n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$.

The paper is organized as follows. Section 2 overviews some known parallel algorithms and tools that are used by the new algorithm. Section 3 presents a simple version of the algorithm that tests if a string is square-free and Section 4 develops a more complicated version that finds all the squares. Section 5 is devoted to the lower bound and Section 6 gives tight bounds for any given number of processors. Concluding remarks are given in Section 7.

2 The CRCW-PRAM model

The algorithms described in this paper are for the concurrent-read concurrent write parallel random access machine model. We use the weakest version of this model called the *common CRCW-PRAM*. In this model, many processors have access to a shared memory. Concurrent read and write operations are allowed at all memory locations. If few processors attempt to write simultaneously to the same memory location, then they all write the same value.

The square detection algorithm uses a string matching algorithm. The input to the string matching algorithm consists of two strings, $pattern[1..m]$ and $text[1..n]$, and the output is a Boolean array $match[1..n]$ that has a '*true*' value at each position where an occurrence of the pattern starts in the text. We use Breslauer and Galil's [10] parallel string matching algorithm that takes $O(\log\log n)$ time using an $n/\log\log n$-processor CRCW-PRAM. This algorithm is the fastest optimal parallel string matching algorithm on general alphabets as shown by Breslauer and Galil [11]. If p processors are available, then the time bounds for the string matching problem are $\Theta(\lceil n/p \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$.

The square detection algorithm also uses an algorithm of Fich, Ragde and Wigderson [19] to compute the minima of n integers in the range $1, \cdots, n$, in constant time using an n-processor CRCW-PRAM. This minima algorithm, for example, can find the first occurrence of a string in another string: after the occurrences are computed by the string matching algorithm mentioned above, look for the smallest i such that $match[i] =$ '*true*'.

Finally, we use the following theorem:

Theorem 2.1 *(Brent [8]) Any parallel algorithm of time t that consists of a total of x elementary operations can be implemented on p processors in $\lceil x/p \rceil + t$ time.*

If we get back to the example above, which finds the first occurrence of one string in an other, we see that the second step of finding the smallest index of an occurrence takes constant time using n processors, while the use of the string matching procedure takes $O(\log\log n)$ time using $n/\log\log n$ processors. By Theorem 2.1 the second step can be slowed down to work in $O(\log\log n)$ time using $n/\log\log n$ processors.

3 Testing if a string is square-free

This section describes an algorithm that tests if a string $\mathcal{S}[1..n]$ is square-free. The algorithm that finds all squares is more involved and is given in the Section 4.

Theorem 3.1 *There exists an algorithm that tests if a string $\mathcal{S}[1..n]$ over a general alphabet is square-free in $O(\log\log n)$ time using $n\log n/\log\log n$ processors.*

Proof: The algorithm consists of independent stages which are computed simultaneously. In stage number η, $0 \le \eta \le \lceil \log_2 n \rceil - 1$, the algorithm looks only for repetitions xx, such that $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$ and $l_\eta = 2^\eta$. If some repetition is found, then a global variable is set to indicate that the string is not square-free. Notice that the complete range of possible lengths of x is covered and if there exist a repetition it will be discovered.

We show how to implement stage number η in $T_\eta = O(\log\log l_\eta)$ time and $O(n)$ operations. Since there are $O(\log n)$ stages, the total number of operations is $O(n\log n)$. By Theorem 2.1, the algorithm can be implemented in $\max T_\eta = O(\log\log n)$ time using $n\log n/\log\log n$ processors. $\Box$

3.1 The stages

We describe stage number η, $0 \le \eta \le \lceil \log_2 n \rceil - 1$, that looks only for repetitions xx, such that $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$. To simplify the presentation, assume without loss of generality that the algorithm can access symbols whose indices are out of the boundaries of the input string. Comparisons to such symbols are answered as unequal.

Partition the input string $\mathcal{S}[1..n]$ into consecutive blocks of length l_η. That is, block number k, for $1 \le k < \lfloor n/l_\eta \rfloor$, is $\mathcal{S}[(k-1)l_\eta + 1..kl_\eta]$. Let $\mathcal{B} = \mathcal{S}[\mathcal{P}..\mathcal{P} + l_\eta - 1]$ be one of these blocks. A repetition xx is said to be *hinged* on $\mathcal{B}$ if $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$ and $\mathcal{B}$ is fully contained in the first copy of x. Stage number η consists of sub-stages which are also computed simultaneously. There is a sub-stage for each block of length l_η. Each sub-stage checks if there is any repetition which is hinged on the block that it is assigned to.

The sub-stage which is assigned to the block $\mathcal{B}$ starts with a call to the string matching algorithm to find all occurrences of $\mathcal{B}$ in $\mathcal{S}[\mathcal{P} + 2l_\eta - 1..\mathcal{P} + 5l_\eta - 3]$. Let $p_1 < p_2 < \cdots < p_r$ be the indices of these occurrences. Then $\mathcal{P} + 2l_\eta - 1 \le p_i < \mathcal{P} + 4l_\eta - 1$, for $i = 1, \cdots, r$. See Figure 1.

Notice that for each repetition xx that is hinged on $\mathcal{B}$ there must be an occurrence of $\mathcal{B}$ at position $\mathcal{P} + |x|$. This occurrence is included in the $\{p_i\}$ sequence.

Lemma 3.2 *For each p_i, one can test in constant time and $O(l_\eta)$ operations if there is any repetition xx that is hinged on $\mathcal{B}$, such that $|x| = p_i - \mathcal{P}$.*

Proof: Let $l = p_i - \mathcal{P}$. We are looking for repetitions xx, such that $|x| = l$. For all ζ in the range $\mathcal{P} + l_\eta - 1 \le \zeta \le p_i$ check if $\mathcal{S}[\zeta - l] = \mathcal{S}[\zeta]$ and if $\mathcal{S}[\zeta] = \mathcal{S}[\zeta + l]$.

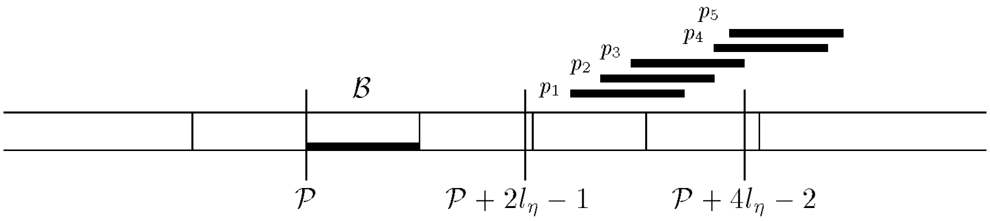

Figure 1: The sub-stage which is assigned to the block $\mathcal{B}=\mathcal{S}[\mathcal{P}..\mathcal{P}+l_\eta-1]$ finds all occurrences of $\mathcal{B}$ that start between positions $\mathcal{P}+2l_\eta-1$ and $\mathcal{P}+4l_\eta-2$.

Let ζ_L be the largest index in this range such that $\mathcal{S}[\mathcal{P}+l_\eta..\zeta_L]=\mathcal{S}[\mathcal{P}+l_\eta+l..\zeta_L+l]$ and ζ_R be the smallest index such that $\mathcal{S}[\zeta_R..p_i-1]=\mathcal{S}[\zeta_R-l..\mathcal{P}-1]$. One can find ζ_L and ζ_R in constant time and $O(l_\eta)$ operations using the integer minima algorithm of Fich, Ragde and Wigderson [19].

We show that there are repetitions xx that are hinged on $\mathcal{B}$, such that $|x|=l$, if and only if $\zeta_R\le\zeta_L+1$. Moreover, these repetitions start at positions s, for $\zeta_R-l\le s\le\zeta_L-l+1$.

If there is a repetition xx that is hinged on $\mathcal{B}$ starting at position s, such that $|x|=l$, then $\mathcal{S}[\zeta-l]=\mathcal{S}[\zeta]$, for all ζ in the range $s+l\le\zeta<p_i$, and $\mathcal{S}[\zeta]=\mathcal{S}[\zeta+l]$, for all ζ in the range $\mathcal{P}+l_\eta\le\zeta<s+l$. But then, $\zeta_L\ge s+l-1$ and $\zeta_R\le s+l$, and thus $\zeta_R-l\le s\le\zeta_L-l+1$ and $\zeta_R\le\zeta_L+1$. See Figure 2.

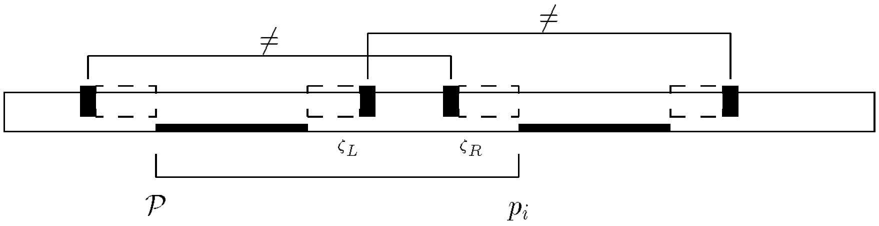

Figure 2: If $\zeta_R>\zeta_L+1$, then there is no repetition xx that is hinged on the block $\mathcal{B}$, such that $|x|=p_i-\mathcal{P}$.

On the other hand, if $\zeta_R\le\zeta_L+1$, then $\mathcal{S}[\zeta_R-l..\zeta_L]=\mathcal{S}[\zeta_R..\zeta_L+l]$. (Recall that there is an occurrence of $\mathcal{S}[\mathcal{P}..\mathcal{P}+l_\eta-1]$ at position p_i and thus $\mathcal{S}[\mathcal{P}..\mathcal{P}+l_\eta-1]=\mathcal{S}[p_i..p_i+l_\eta-1]$.) The last equality means that there are repetitions xx, such that $|x|=l$, starting at positions s, for $\zeta_R-l\le s\le\zeta_L-l+1$. $\square$

The algorithm can check if any of the p_i's corresponds to a repetition in constant time using Lemma 3.2, but it would make $O(rl_\eta)$ operations if the length of the $\{p_i\}$ sequence is r. Luckily, for now, the algorithm has only to test if the string is square-free and it does not have to check if all the p_i's correspond to repetitions; if $r > 2$, then $\mathcal{S}[1..n]$ must contain a square as the following lemma shows.

Lemma 3.3 *If the length of the $\{p_i\}$ sequences is $r > 2$, then $\mathcal{S}[1..n]$ contains a repetition. This repetition is shorter than the repetitions that are supposed to be found in this stage.*

Proof: Recall that $\mathcal{P} + 2l_\eta - 1 \leq p_i < \mathcal{P} + 4l_\eta - 1$, for $i = 1, \cdots, r$. If $r \geq 3$, then either $p_2 - p_1 \leq l_\eta$ or $p_3 - p_2 \leq l_\eta$. But then, there is a repetition xx, such that $|x| = p_2 - p_1$ or $|x| = p_3 - p_2$ (respectively), starting at position p_1 or p_2 (respectively). $\square$

The computation in each sub-stage of stage η can be summarized as follows:

1. Compute the $\{p_i\}$ sequence.
2. If the $\{p_i\}$ sequence has more than two elements, then by Lemma 3.3, the string $\mathcal{S}[1..n]$ contains a repetition. This repetition will also be found by some stage number μ, $\mu < \eta$.
3. If the $\{p_i\}$ sequence has at most two elements, check if these elements correspond to repetitions using the procedure described in Lemma 3.2.

Lemma 3.4 *Stage number η is correct. It takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations.*

Proof: For correctness we have to show that if the string $\mathcal{S}[1..n]$ contains any repetition xx, such that $2l_\eta - 1 \leq |x| < 2l_{\eta+1} - 1$, then some repetition will be found. Assume that there is such a repetition. Since $2l_\eta - 1 \leq |x|$, there must be a block of length l_η that is completely contained in the first x. The sub-stage which is assigned to that block will either find the repetition xx or conclude that there is a shorter repetition by Lemma 3.3. In both cases some repetition has been found. Notice that some repetitions can be detected by several stages and sub-stages simultaneously.

Stage number η consists of $\lfloor n/l_\eta \rfloor$ independent sub-stages. In each sub-stage, step number 1 takes $O(\log\log l_\eta)$ time and $O(l_\eta)$ operation using Breslauer and Galil's string matching algorithm. Steps number 2 and 3 take constant time and make $O(l_\eta)$ operations. Since all the sub-stages are computed in parallel, stage number η takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations. $\square$

4 Detecting all squares

In this section we show how the algorithm that was given in Section 3 can be generalized to find all squares in a string.

Beame and Hastad [5] proved a lower bound of $\Omega(\log n / \log\log n)$ time for computing the parity of n input bits on CRCW-PRAMs with any polynomial number of processors. This lower bound implies that many "interesting" problems would require at least that time. However, several string problems, including the problem of detecting all squares in a string, have constant time solutions using polynomial number of processors.

While the problem of testing if a string is square-free has only a single output bit, the problem of finding all squares has a more complicated output structure. If we wish to obtain algorithms that get around Beame and Hastad's lower bound we can not count the number of squares that are found and therefore we can not list them contiguously in an array. Instead we will represent the output of the algorithm in a sparse array with $O(n \log n)$ entries. Notice that this problem did not exist in the previous square detection algorithms since their time bounds were at least $O(\log n)$.

Similar to the testing algorithm, the square detection algorithm proceeds in independent stages which are computed within the same time and processor bounds as before. Only now, since the algorithm must find all the squares, the following difficulties arise.

1. The detection algorithm can not use Lemma 3.3 only to conclude that the string is not square-free; it must find all the squares.

2. The algorithm has to verify which repetitions are squares. This was not necessary before since a string is square-free if and only if it is repetition-free.

3. The squares have to be represented in a sparse array with $O(n \log n)$ entries.

The first two issues will be addressed in Section 4.1 that describes the stages of the square detection algorithm, while the third issue is discussed next.

The following lemma is used to justify the output representation used by the algorithm.

Lemma 4.1 *(see, e.g., Crochemore and Rytter [18]) If there are three squares xx, yy and zz, such that $|x| < |y| < |z|$, that start at the same position of some string, then $|x| + |y| \leq |z|$.*

Recall that in stage number η the algorithm looks only for squares xx, such that $2l_\eta - 1 \leq |x| < 2l_{\eta+1} - 1$ and $l_\eta = 2^\eta$. Therefore, by Lemma 4.1, there are no more than two squares that start at each position of the input string and have to be discovered in the same stage. Thus, the output can be represented in an array that will hold, for each position of the input string and for each stage, the two squares that might be detected starting at the specific position in the specific stage. (e.g., let u be primitive and v a non-empty proper prefix of u. Then the string $u^k vu^{k+1}vu$, $k \geq 1$, contains the two prefix squares $u^k vu^k v$ and $u^k vuu^k vu$ whose lengths differ by $2|u|$. If $k \geq 2$, then it contains also the prefix square uu, and if $k = 2$, then the inequality in Lemma 4.1 is tight. In the extreme case, by letting $u =$ 'ab' and $v =$ 'a', one gets arbitrary long pairs of squares whose lengths differ by 4.)

The complexity bounds of the square detection algorithm are summarized in the following theorem.

Theorem 4.2 *There exists an algorithm that finds all squares in a string $\mathcal{S}[1..n]$ over a general alphabet in $O(\log\log n)$ time using $n \log n/ \log\log n$ processors.*

4.1 The stages

Consider a single stage. As in Section 3.1, the input string $\mathcal{S}[1..n]$ is partitioned into consecutive blocks of length l_η and there is a sub-stage that is assigned to each such block. To simplify the presentation we allow squares to be discovered by several sub-stages simultaneously: the sub-stage that is assigned to the block $\mathcal{B}$ discovers all the squares which are hinged on this block. Later, we make sure that the information about each square is written only once into the output array by reporting only those squares for which $\mathcal{B}$ is the leftmost block fully contained in the square. Thus, stage number η finds all squares xx, such that $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$.

As already noted, each square that is hinged on $\mathcal{B}$ ties the block $\mathcal{B}$ to a specific replica. The sub-stage that is assigned to $\mathcal{B}$ starts with a call to the string matching algorithm to find the viable replicas of $\mathcal{B}$. Let $p_1 < \cdots < p_r$ denote their indices.

Definition 4.3 *A string x is a* rotation *of another string $\hat{x}$ (and vice versa) if $x = uv$ and $\hat{x} = vu$ for some strings u and v.*

Definition 4.4 *A string $\mathcal{S}$ has* a period *u if $\mathcal{S}$ is a prefix of u^k for some large enough k. Alternatively, a string $\mathcal{S}[1..n]$ has a period of length π if $\mathcal{S}[i] = \mathcal{S}[i+\pi]$, for $i = 1, \cdots, n-\pi$. The shortest period of a string $\mathcal{S}$ is called* the period *of $\mathcal{S}$.*

Lemma 4.5 *(Lyndon and Schutzenberger [24]) If a string of length m has two periods of lengths p and q, and $p + q \le m$, then it also has a period of length $gcd(p, q)$.*

The task of the sub-stage is to identify which of the p_i's corresponds to squares that are hinged on $\mathcal{B}$. In Lemma 3.2 we have shown that it is possible to verify efficiently that some specific p_i corresponds to repetitions xx that are hinged on $\mathcal{B}$, such that $|x| = p_i - \mathcal{P}$. The proof of Lemma 3.2 reveals that those differences $p_i - \mathcal{P}$ that pass the repetition-detection test actually expose an entire sequence of repetitions which are consecutive rotations of the same repetition. Such a sequence will be called a *family* of repetitions.

Lemma 4.6 *A family of repetitions contains a square if and only if all the repetitions in the family are squares.*

Proof: Let xx be a repetition but not a square. Thus $x = z^l$ and $l > 1$. If $\hat{x}$ is a rotation of x, then $\hat{x} = v(uv)^j(uv)^{l-j-1}u = (vu)^l$ where $z = uv$, and thus $\hat{x}$ is not primitive. □

The last lemma means that if we wish to certify that repetitions are actually squares it is enough to certify one repetition in each family. The next lemma shows how to test efficiently that a given repetition is indeed a square by solving a single string matching problem. (The technique for primitive certification proposed by Apostolico [2] uses information about shorter squares which are discovered in other stages. We use a different method that keeps the stages in the algorithm completely independent.)

Lemma 4.7 *Given a repetition xx, let l be the index of the first occurrence of x in xx, other than the trivial occurrence at the beginning of xx. Then, xx is a square if and only if $l = |x|$.*

Proof: Clearly, $l \le |x|$. If $x = z^j$, then $xx = z^{2j}$ and x occurs at position $|z|$ of xx. On the other hand, if $l < |x|$, then xx has periods of lengths l and $|x|$ and by Lemma 4.5, l divides $|x|$. But then $x = z^{|x|/l}$ is not primitive. □

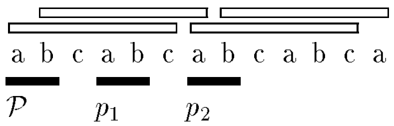

Figure 3: Repetitions must be certified to be squares. In this example, the repetitions in the family that corresponds to $p_2 - \mathcal{P}$ are not squares.

Given a replica of $\mathcal{B}$ at position p_i, we can find the family of repetitions xx, such that $|x| = p_i - \mathcal{P}$, using Lemma 3.2, and then we can certify that these repetitions are actually squares using Lemma 4.7. See Figure 3.

However, if the length of the $\{p_i\}$ sequence is large, then repeating the process above for each p_i can be costly. Moreover, it is a problem even to find and to manipulate the $\{p_i\}$ sequence efficiently. The following lemmas will help to overcome this difficulty.

Lemma 4.8 *Assume that the period length of a string $W[1..l]$ is p. If $W[1..l]$ occurs only at positions $p_1 < p_2 < \cdots < p_k$ of a string V and $p_k - p_1 \le \lceil l/2 \rceil$, then the p_i's form an arithmetic progression with difference p.*

Proof: Assume $k \ge 2$. We prove that $p = p_{i+1} - p_i$ for $i = 1, \cdots, k-1$. The string W has periods of lengths p and $q = p_{i+1} - p_i$. Since $p \le q \le \lceil l/2 \rceil$, by Lemma 4.5 it also has a period of length $gcd(p, q)$. But p is the length of the shortest period so $p = gcd(p, q)$ and p must divide q. The string $V[p_i..p_{i+1}+l-1]$ has period of length p. If $q > p$, then there must be another occurrence of W at position $p_i + p$ of V; a contradiction. □

Recall that $\mathcal{P} + 2l_\eta - 1 \le p_i < \mathcal{P} + 4l_\eta - 1$. To utilize the last lemma it is convenient to partition the sequence $\{p_i\}$ and to regard the sub-stage as consisting of four consecutive *phases*. Each phase handles viable replicas of $\mathcal{B}$ in a sub-block of size $l_\eta/2$ (hereafter, a $l_\eta/2$-block). We describe a generic phase, involving the occurrences of $\mathcal{B}$ at positions $q_1 < \cdots < q_k$, where $\{q_i\}$ is a sub-sequence of $\{p_i\}$ that lists all the occurrences that fall within a $l_\eta/2$-block. (In the first stages there are fewer phases.)

Lemma 4.9 *The sequence $\{q_i\}$ of occurrences of $\mathcal{B}$ in a $l_\eta/2$-block is an arithmetic progression with difference q, where q is the period length of $\mathcal{B}$.*

Proof: An immediate consequence of Lemma 4.8. □

The sequence $\{q_i\}$ can be represented using three integers: the start, the difference and the sequence length. This representation can be easily computed from the output of the string matching algorithm (which is a Boolean vector) using Fich, Ragde and Wigderson's [19] integer minima algorithm in constant time using $O(l_\eta)$ operations. This idea has been successfully applied also in efficient parallel algorithms for other string problems [3, 9, 12].

If the $\{q_i\}$ sequence does not contain any elements, then the phase does not need to do anything. If there is one element q_1, then the algorithm finds the family of repetitions that are associated with the difference $q_1 - \mathcal{P}$ and certifies them to be squares as described above. The next lemmas are used in phases that have longer $\{q_i\}$ sequences.

Assume that the length of the arithmetic progression $\{q_i\}$ is $k \geq 2$ and let q be the difference of the progression. By Lemmas 4.8 and 4.9, the block $\mathcal{B} = \mathcal{S}[\mathcal{P}..\mathcal{P}+l_\eta-1]$ and the sub-string covered by the occurrences of this block at positions q_i, $\mathcal{S}[q_1..q_k+l_\eta-1]$, have period length q. The algorithm proceeds by checking how far this periodicity extends on both sides of these sub-strings.

Let α_L and α_R be the positions where the periodicity of length q terminates on the left and on the right of $\mathcal{B}$, respectively, and let γ_L and γ_R be the positions where the periodicity of length q terminates on the left and on the right of the sub-string $\mathcal{S}[q_1..q_k+l_\eta-1]$, respectively. We are interested in these indices only if $\mathcal{P}-(q_k-\mathcal{P})+l_\eta \leq \alpha_L$, $\alpha_R < q_1+l_\eta$, $\mathcal{P} \leq \gamma_L$ and $\gamma_R < 2q_k - \mathcal{P}$, and these indices are undefined otherwise. Namely, if all indices are defined, then $\mathcal{S}[\alpha_L+1..\alpha_R-1]$ has period length q, $\mathcal{S}[\alpha_L] \neq \mathcal{S}[\alpha_L+q]$, $\mathcal{S}[\alpha_R] \neq \mathcal{S}[\alpha_R-q]$,

$$\mathcal{P}-(q_k-\mathcal{P})+l_\eta \leq \alpha_L < \mathcal{P} \quad \text{and} \quad \mathcal{P}+l_\eta \leq \alpha_R < q_1+l_\eta,$$

$\mathcal{S}[\gamma_L+1..\gamma_R-1]$ has period length q, $\mathcal{S}[\gamma_L] \neq \mathcal{S}[\gamma_L+q]$, $\mathcal{S}[\gamma_R] \neq \mathcal{S}[\gamma_R-q]$ and

$$\mathcal{P} \leq \gamma_L < q_1 \quad \text{and} \quad q_k+l_\eta \leq \gamma_R < 2q_k-\mathcal{P}.$$

It is possible to compute the indices α_L, α_R, γ_L and γ_R, or to decide which indices are undefined, in constant time and $O(l_\eta)$ operations using Fich, Ragde and Wigderson's [19] integer minima algorithm.

The following lemmas classify the possible interactions between α_L, α_R, γ_L and γ_R and their effect on the squares that are hinged on $\mathcal{B}$.

Lemma 4.10 *If one of α_R and γ_L is defined, then so is the other one, and $\alpha_R - \gamma_L \leq q$.*

Proof: By the definition of α_R and γ_L, $\mathcal{S}[\mathcal{P}..\alpha_R-1]$ and $\mathcal{S}[\gamma_L+1..q_k+l_\eta-1]$ have period length q, $\mathcal{S}[\alpha_R] \neq \mathcal{S}[\alpha_R-q]$ and $\mathcal{S}[\gamma_L] \neq \mathcal{S}[\gamma_L+q]$.

If $q < \alpha_R - \gamma_L$, then by the periodicity of $\mathcal{S}[\mathcal{P}..\alpha_R-1]$ and since $\gamma_L + q < \alpha_R$, we get that $\mathcal{S}[\gamma_L] = \mathcal{S}[\gamma_L+q]$, in contradiction to the definition of α_R and γ_L. Therefore, $\alpha_R - \gamma_L \leq q$ or at least one of α_R and γ_L is undefined.

If α_R is undefined, then $\mathcal{S}[\mathcal{P}..q_1+l_\eta-1]$ has period length q and the argument above shows that γ_L can not be defined. The proof of the symmetric case is identical. $\square$

The following lemma identifies certain repetitions that can never be squares.

Lemma 4.11 *If both α_R and γ_L are undefined, then none of the repetitions possibly hinged on $\mathcal{B}$ is a square.*

Proof: If α_R and γ_L are undefined, then $\mathcal{S}[\mathcal{P}..q_k + l_\eta - 1]$ has period length q. Consider any q_i and let $l = q_i - \mathcal{P}$. By the periodicity above and since $\mathcal{S}[\mathcal{P}..\mathcal{P} + l_\eta - 1] = \mathcal{S}[q_i..q_i + l_\eta - 1]$, we get that $\mathcal{S}[\mathcal{P}..\mathcal{P} + q_k - q_i + l_\eta - 1] = \mathcal{S}[q_i..q_k + l_\eta - 1]$. Thus, the sub-string $\mathcal{S}[\mathcal{P}..q_k + l_\eta - 1]$ has a period of length l. But $q \leq l_\eta/2 < l$ and by Lemma 4.5, q divides l.

Let xx be a repetition that is hinged on $\mathcal{B}$ starting at position s, such that $|x| = l$. Then $x = \mathcal{S}[s..s + l - 1] = \mathcal{S}[s + l..\mathcal{P} + l - 1]\mathcal{S}[\mathcal{P}..s + l - 1]$ has period length q and therefore x is not primitive. $\square$

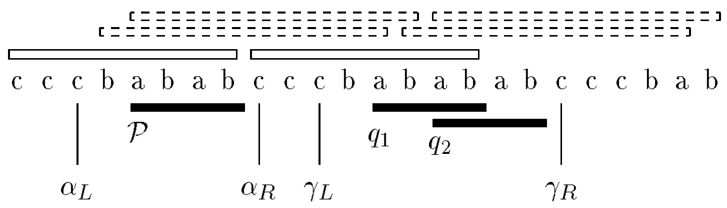

Figure 4: There can be at most two families of synchronized squares. In this example, one family corresponds to $\gamma_L - \alpha_L = q_1 - \mathcal{P}$ and the other to $\gamma_R - \alpha_R = q_2 - \mathcal{P}$.

If both α_R and γ_L are defined, then certain repetitions, which are characterized in the next lemma, must align α_R with γ_R and α_L with γ_L. These repetitions are called *synchronized repetitions.* See Figure 4.

It is convenient to state the next lemmas in terms of the positions where the repetitions are centered; a repetition xx that starts at position s is *centered* at position $s + |x|$.

Lemma 4.12 *If both α_R and γ_L are defined then:*

1. *Repetitions that are hinged on $\mathcal{B}$ and centered at positions h, such that $h \leq \gamma_L$, may exist only if α_L is defined. These repetitions constitute a family of repetitions that corresponds to the difference $q_i - \mathcal{P}$, provided that there exists some q_i such that $\gamma_L - \alpha_L = q_i - \mathcal{P}$.*

2. *Repetitions that are hinged on $\mathcal{B}$ and centered at positions h, such that $\alpha_R < h$, may exist only if γ_R is defined. These repetitions constitute a*

family of repetitions that corresponds to the difference $q_j - \mathcal{P}$, provided that there exists some q_j such that $\gamma_R - \alpha_R = q_j - \mathcal{P}$.

Notice that if $\alpha_R < \gamma_L$, then repetitions whose center h satisfies $\alpha_R < h \leq \gamma_L$ may exist only if both α_L and γ_R are defined and if $\gamma_R - \alpha_R = \gamma_L - \alpha_L$.

Proof: Let $xx = \mathcal{S}[h - l..h + l - 1]$ be a repetition that is hinged on $\mathcal{B}$ and centered at position h, such that $|x| = q_i - \mathcal{P}$, and let $l = |x|$.

Assume $\mathcal{P} + l_\eta \leq h \leq \gamma_L$. The proof distinguishes between two cases. If α_L is undefined or if $\alpha_L < \gamma_L - l$ (see Figure 5), then by the periodicity in the definition of α_L and γ_L, $\mathcal{S}[\gamma_L - l + q] = \mathcal{S}[\gamma_L + q]$ and $\mathcal{S}[\gamma_L - l] = \mathcal{S}[\gamma_L - l + q]$. Since there is the repetition xx, also $\mathcal{S}[\gamma_L - l] = \mathcal{S}[\gamma_L]$. Thus $\mathcal{S}[\gamma_L] = \mathcal{S}[\gamma_L + q]$ in contradiction to the fact that $\mathcal{S}[\gamma_L] \neq \mathcal{S}[\gamma_L + q]$ by the definition of γ_L.

Similarly, if $\alpha_L > \gamma_L - l$, then by the periodicity in the definition of α_L and γ_L, $\mathcal{S}[\alpha_L + q] = \mathcal{S}[\alpha_L + l + q]$ and $\mathcal{S}[\alpha_L + l] = \mathcal{S}[\alpha_L + l + q]$. Since there is the repetition xx, also $\mathcal{S}[\alpha_L] = \mathcal{S}[\alpha_L + l]$. Thus, $\mathcal{S}[\alpha_L] = \mathcal{S}[\alpha_L + q]$ in contradiction to the fact that $\mathcal{S}[\alpha_L] \neq \mathcal{S}[\alpha_L + q]$ by the definition of α_L.

Therefore, such a repetition xx may exist only if $\alpha_L = \gamma_L - l$, or in other words if $\gamma_L - \alpha_L = q_i - \mathcal{P}$ for some q_i. Since α_L, γ_L and $\mathcal{P}$ are given, there is at most one such q_i.

The proof of the second part where $\alpha_R < h$ is similar. □

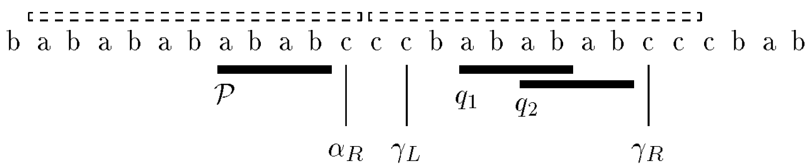

Figure 5: Illustrating one of the cases in Lemma 4.12: an undefined α_L is incompatible with any repetition hinged on $\mathcal{B}$ and centered at a position $h \leq \gamma_L$.

As a consequence of the last lemma, there can be at most two repetition families (in each phase) that have to be verified and certified to be squares. However, there are squares which might have been missed since Lemma 4.12 did cover all eventualities. If $\gamma_L < \alpha_R$, then there might exist repetitions whose center h satisfies $\gamma_L < h \leq \alpha_R$. These repetitions are called *unsynchronized repetitions*. We classify these repetitions next and show that if such repetitions exist, then they must be squares.

Lemma 4.13 *If α_R and γ_L are defined and $\gamma_L < \alpha_R$, then there might be a family of repetitions associated with each of the differences $l = q_i - \mathcal{P}$, with* *centers at positions h, such that $\gamma_L < h \le \alpha_R$. The repetitions in each such family are all squares, and they are centered at positions h, such that $\max(\alpha_L + l, \gamma_L) < h \le \min(\alpha_R, \gamma_R - l)$. Notice that such a family is not empty if and only if $l < \min(\alpha_R - \alpha_L, \gamma_R - \gamma_L)$.*

Proof: Consider repetitions $\mathcal{S}[h-l..h-1] = \mathcal{S}[h..h+l-1]$ that are associated with the difference $l = q_i - \mathcal{P}$ and whose centers h satisfy $\gamma_L < h \le \alpha_R$. We show that such repetitions exist if and only if $\alpha_L + l < h$ and $h \le \gamma_R - l$. (Ignoring the constraints involving undefined indices.)

If $h \le \alpha_L + l$, then $\mathcal{S}[\alpha_L] = \mathcal{S}[\alpha_L + l]$. Since $\gamma_L < h$, we know that $\mathcal{S}[\alpha_L + l] = \mathcal{S}[\alpha_L + l + q]$. But then, $\mathcal{S}[\alpha_L] = \mathcal{S}[\alpha_L + q]$, in contradiction to the definition of α_L. Similarly, it is impossible that $\gamma_R - l < h$.

On the other hand, if $\max(\alpha_L + l, \gamma_L) < h \le \min(\alpha_R, \gamma_R - l)$, then $\mathcal{S}[h-l..h-1]$ and $\mathcal{S}[h..h+l-1]$ have period length q. Since $\mathcal{S}[\mathcal{P}..\mathcal{P}+l_\eta-1] = \mathcal{S}[q_i..q_i+l_\eta-1]$ we get that $\mathcal{S}[h-l..h-1] = \mathcal{S}[h..h+l-1]$. (The same reasoning holds also if α_L or γ_R are not defined.)

It remains to show that these repetitions are actually squares. If $\mathcal{S}[h-l..h-1] = z^j$ for some $j > 1$, then $\mathcal{S}[h-l..h-1]$ has periods of length q and $|z|$ and by Lemma 4.5, q divides $|z|$. But then, $\mathcal{S}[h-q..h-1] = \mathcal{S}[h..h+q-1]$ and $\alpha_R - \gamma_L \ge 2q$, in contradiction to Lemma 4.10. □

The computation in each sub-stage of the square detection algorithm can be summarized as follows:

1. Compute the $\{p_i\}$ sequence and proceed in four phases.
2. In each phase, find the arithmetic progression $\{q_i\}$.
3. If the $\{q_i\}$ sequence has a single element q_1, then find the repetition family that corresponds to q_1 using Lemma 3.2 and certify that these repetitions are squares using Lemma 4.7.
4. If the $\{q_i\}$ sequence has at least two elements, then:
   (a) Find the synchronized repetition families using Lemma 4.12 and certify that these repetitions are squares using Lemma 4.7.
   (b) Find the unsynchronized squares using Lemma 4.13.

Lemma 4.14 *Stage number η is correct. It takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations.*

Proof: It is clear that if the string $\mathcal{S}[1..n]$ contains any square xx, such that $2l_\eta - 1 \le |x| < 2l_{\eta+1} - 1$, then there must be a block $\mathcal{B}$ of length l_η that is the leftmost block completely contained in the square. We have seen that the sub-stage that is assigned to the block $\mathcal{B}$ will find xx.

Stage number η consists of $\lfloor n/l_\eta \rfloor$ independent sub-stages. Each sub-stage might make at most nine calls to Breslauer and Galil's string matching algorithm: one to find the $\{p_i\}$ sequence and at most two in each phase to certify squares using Lemma 4.7. These calls take $O(\log\log l_\eta)$ time and make $O(l_\eta)$ operations. The rest of the work in each sub-stage takes constant time and $O(l_\eta)$

operations. Since all the sub-stages are computed in parallel, stage number η takes $O(\log\log l_\eta)$ time and makes $O(n)$ operations. □

Remark. Assume that the sequence $\{q_i\}$ has $k > 1$ elements and difference q. If α_R and γ_L are defined, then some synchronizing repetitions might have to be certified to be squares. It easy to check that for the repetitions xx that arise in this case, if $x = z^j$, then $j \le \epsilon$ for some small positive constant ϵ. Thus, it is sufficient to verify that $x \ne z^j$, for $j = 2, \cdots, \epsilon$, in order to certify that x is primitive. This is more efficient than the general square certification method suggested in Lemma 4.7.

5 The lower bound

We prove a lower bound for testing if a string is square-free by a reduction to Breslauer and Galil's [11] lower bound for string matching. Breslauer and Galil show that an adversary can fool any algorithm which claims to check if a string has a period that is shorter than half of its length in fewer than $\Omega(\lceil n/p \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ rounds with p comparisons in each round. The lower bound holds for the CRCW-PRAM model in the case of general alphabets where the only access an algorithm has to the input string is by pairwise symbol comparisons.

We will not report the details of that lower bound. We only use the fact that the adversary generates a string $\mathcal{S}[1..n]$ that has the following property: If $\mathcal{S}[i] = \mathcal{S}[j]$, then $\mathcal{S}[k] = \mathcal{S}[i]$, for any integer k, such that $k \equiv i \pmod{|j-i|}$ and $1 \le k \le n$.

Lemma 5.1 *The string generated by Breslauer and Galil's adversary has a period that is shorter than half of its length if and only if it contains a square.*

Proof: If the string generated by the adversary has a period which is shorter than half of its length, then it contains a square that starts at the beginning of the string.

On the other hand, assume that a square xx starts at position s of $\mathcal{S}[1..n]$. Namely, $\mathcal{S}[s+k] = \mathcal{S}[s+|x|+k]$ for $k = 0, \cdots, |x|-1$. But then, by the property mentioned above, the string generated by the adversary has a period of length $|x|$, which is smaller than half of the string length. □

Now, we are ready to prove the lower bound.

Theorem 5.2 *Any parallel algorithm that tests if a string $\mathcal{S}[1..n]$ over general alphabets is square-free must take $\Omega(\lceil \frac{n\log n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ rounds with p comparisons in each round.*

Proof: Main and Lorentz [26] show that any sequential algorithm that tests if a string over general alphabets is square-free must make $\Omega(n \log n)$ comparisons. This gives an immediate lower bound of $\Omega(\lceil \frac{n\log n}{p} \rceil)$ rounds with p comparisons in each round.

By Lemma 5.1, the string that is generated by the adversary of Breslauer and Galil has a period that is shorter than half of its length if and only if it contains a square. Breslauer and Galil show that after $\Omega(\log\log_{\lceil 1+p/n\rceil} 2p)$ rounds the adversary still has the choice of forcing the string to have a period that is shorter than half of its length or not to have any such period. Therefore, any algorithm that tries to decide in fewer rounds if a string is square-free can be fooled. By combining these two bounds we get the claimed lower bound. $\Box$

Corollary 5.3 *Any optimal parallel algorithm that tests if a string* $\mathcal{S}[1..n]$ *is square-free must take* $\Omega(\log\log n)$ *rounds.*

Proof: By Theorem 5.2, the lower bound is $\Omega(\log\log n)$ even with $n\log n$ comparisons in each round. $\Box$

6 The number of processors

This section derives tight bounds for any given number of available processors.

Theorem 6.1 *If* p *processors are available, then the lower and upper bounds for testing if a string is square-free and for detecting all squares are* $\Theta(\lceil\frac{n\log n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$.

Proof: The lower bound was given in Theorem 5.2. It remains to prove the upper bound.

1. If $p \le \frac{n\log n}{\log\log n}$, then by Theorem 2.1, the optimal algorithms of Sections 3 and 4 can be slowed down to run in $O(\frac{n\log n}{p})$ time, matching the lower bound.

2. If $\frac{n\log n}{\log\log n} < p \le n\log n$, then the lower bound is $\Omega(\log\log n)$, matching the time bound of the algorithms with only $\frac{n\log n}{\log\log n}$ processors.

If $p > n\log n$, then we must go back to the algorithms given in Sections 3 and 4. The processors are distributed equally among the stages. In stage number η, the processors are distributed equally among the sub-stages, giving $\frac{p}{n\log n}l_\eta$ processors to each sub-stage.

Since sub-stages that handle strings of length $O(l_\eta)$ have more than l_η processors available, the sub-stages take constant time except for the calls to Breslauer and Galil's string matching algorithm. These calls take $T_\eta = O(\log\log_{\lceil 1+p/n\log n\rceil} 2\frac{p}{n\log n}l_\eta)$ time. Thus, the whole algorithm takes $\max T_\eta = O(\log\log_{\lceil 1+p/n\log n\rceil} \frac{2p}{\log n})$ time.

3. If $p > n\log n$, then it can be easily verified that $\log\log_{\lceil 1+p/n\log n\rceil} \frac{2p}{\log n} \in O(\log\log_{\lceil 1+p/n\rceil} 2p)$, establishing that the lower and upper bounds are the same.

4. If $p > n^{1+\epsilon}$ for some fixed $\epsilon > 0$, then the upper bound is $O(1)$. $\Box$

7 Concluding remarks

The algorithm described in this paper uses a string matching procedure as a "black-box" that has a specific input-output functionality, without going into its implementation details. Breslauer and Galil's string matching algorithm is the fastest possible over general alphabets, however, it is unknown at the moment if a faster algorithm exists over constant size alphabets. If such an algorithm exists, it could be used in a faster algorithm for finding squares. Notice that a fast CRCW-PRAM implementation requires the computation of certain functions such as the log function and integral powers within the time and processor bounds. Regardless of the feasibility of such computation, the algorithm that was described in this paper is valid in the parallel comparison decision tree model.

Our parallel square detection algorithm resembles the sequential algorithms of Main and Lorentz [25, 26]. (The testing algorithm is in fact a parallel implementation of the testing algorithm in [26].) Still, the sequential implementation of our parallel algorithm is interesting on its own. By using a time-space-optimal string-matching algorithm, such as the algorithm of Galil and Seiferas [21], we obtain a time-space-optimal algorithm for detecting squares. By using a real-time string-matching algorithm, such as the algorithm of Galil [20], and a careful treatment of periods within the input string, we obtain an on-line square detection algorithm that reports squares as soon as they are formed, while the input string is extended even on both sides, spending $O(\log n)$ time per symbol. No such algorithms were known before.

8 Acknowledgments

We thank Zvi Galil, Roberto Grossi and Kunsoo Park for their valuable comments.

References

[1] A. Apostolico. On context constrained squares and repetitions in a string. *R.A.I.R.O. Informatique theorique*, 18(2):147–159, 1984.

[2] A. Apostolico. Optimal Parallel Detection of Squares in Strings. *Algorithmica*, 8:285–319, 1992.

[3] A. Apostolico, D. Breslauer, and Z. Galil. Optimal Parallel Algorithms for Periods, Palindromes and Squares. In *Proc. 19th International Colloquium on Automata, Languages, and Programming*, number 623 in Lecture Notes in Computer Science, pages 296–307. Springer-Verlag, Berlin, Germany, 1992.

[4] A. Apostolico and F.P. Preparata. Optimal off-line detection of repetitions in a string. *Theoret. Comput. Sci.*, 22:297–315, 1983.

[5] P. Beame and J. Hastad. Optimal bound for decision problems on the CRCW-PRAM. *J. Assoc. Comput. Mach.*, 36(3):643–670, 1989.

[6] D.R. Bean, A. Ehrenfeucht, and G.F. McNulty. Avoidable patterns in strings of symbols. *Pacific J. Math.*, 85:261–294, 1979.

[7] J. Berstel. Sur les mots sans carré définis par un morphism. In *Proc. 6th International Colloquium on Automata, Languages, and Programming*, number 71 in Lecture Notes in Computer Science, pages 16–25. Springer-Verlag, Berlin, Germany, 1979.

[8] R.P. Brent. Evaluation of general arithmetic expressions. *J. Assoc. Comput. Mach.*, 21:201–206, 1974.

[9] D. Breslauer. *Efficient String Algorithmics*. PhD thesis, Dept. of Computer Science, Columbia University, New York, NY, 1992.

[10] D. Breslauer and Z. Galil. An optimal $O(\log\log n)$ time parallel string matching algorithm. *SIAM J. Comput.*, 19(6):1051–1058, 1990.

[11] D. Breslauer and Z. Galil. A Lower Bound for Parallel String Matching. *SIAM J. Comput.*, 21(5):856–862, 1992.

[12] D. Breslauer and Z. Galil. Finding all Periods and Initial Palindromes of a String in Parallel. *Algorithmica*, to appear.

[13] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inform. Process. Lett.*, 12(5):244–250, 1981.

[14] M. Crochemore. Sharp characterizations of squarefree morphisms. *Inform. Process. Lett.*, 18:221–226, 1982.

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

[16] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inform. Process. Lett.*, 38:57–60, 1991.

[17] M. Crochemore and W. Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *Theoret. Comput. Sci.*, 88:59–82, 1991.

[18] M. Crochemore and W. Rytter. Periodic Prefixes in Texts. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Proc. of the Sequences '91 Workshop: "Sequences II: Methods in Communication, Security and Computer Science"*, pages 153–165. Springer-Verlag, Berlin, Germany, 1993.

[19] F.E. Fich, R.L. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17(3):606–627, 1988.

[20] Z. Galil. String Matching in Real Time. *J. Assoc. Comput. Mach.*, 28(1):134–149, 1981.

[21] Z. Galil and J. Seiferas. Time-space-optimal string matching. *J. Comput. System Sci.*, 26:280–294, 1983.

[22] M. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA., U.S.A., 1978.

[23] M. Lothaire. *Combinatorics on Words.* Addison-Wesley, Reading, MA., U.S.A., 1983.

[24] R.C. Lyndon and M.P. Schutzenberger. The equation $a^m = b^n c^p$ in a free group. *Michigan Math. J.*, 9:289–298, 1962.

[25] G.M. Main and R.J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5:422–432, 1984.

[26] G.M. Main and R.J. Lorentz. Linear time recognition of squarefree strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 271–278. Springer-Verlag, Berlin, Germany, 1985.

[27] M.O. Rabin. Discovering Repetitions in Strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 279–288. Springer-Verlag, Berlin, Germany, 1984.

[28] R. Ross and R. Winklmann. Repetitive strings are not context-free. Technical Report CS-81-070, Washington State University, Pullman, WA, 1981.

[29] A. Thue. Über unendliche zeichenreihen. *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, (7):1–22, 1906.

[30] A. Thue. Über die gegenseitige lage gleicher teile gewisser zeichenreihen. *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, (1):1–67, 1912.

Recent Publications in the BRICS Report Series

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

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

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

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

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

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

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

RS-95-4 **Mogens Nielsen and Glynn Winskel.** ***Petri Nets and Bisimulations*****. January 1995. 36 pp. To appear in TCS.**

RS-95-3 **Anna Ingólfsdóttir.** ***A Semantic Theory for Value–Passing Processes, Late Approach, Part I: A Denotational Model and Its Complete Axiomatization*****. January 1995. 37 pp.**

RS-95-2 **François Laroussinie, Kim G. Larsen, and Carsten Weise.** ***From Timed Automata to Logic - and Back*****. January 1995. 21 pp.**

RS-95-1 **Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum.** ***Dynamic Algorithms for the Dyck Languages*****. January 1995. 21 pp.**