1991

Optimal Parallel Algorithms for Periods, Palindromes and Squares (Preliminary Version)

Alberto Apostolico

Dany Breslauer

Zvi Galil

Report Number:
91-082

OPTIMAL PARALLEL ALGORITHMS FOR PERIODS, PALINDROMES AND SQUARES

Alberto Apostolico
Dany Breslauer

CSD-TR-91-082
November 1991

Optimal Parallel Algorithms for Periods, Palindromes and Squares (Preliminary Version)

Alberto Apostolico*
Purdue University and
Università di Padova

Dany Breslauer†
Columbia University

Zvi Galil‡
Columbia University and
Tel-Aviv University

Summary of results

Optimal concurrent-read concurrent-write parallel algorithms for two problems are presented:

- Finding all the periods of a string. The period of a string can be computed by previous efficient parallel algorithms only if it is shorter than half of the length of the string. Our new algorithm computes all the periods, even if they are longer, in optimal $O(\log\log n)$ time. The algorithm can be used to compute all initial palindromes of a string within the same bounds.
- Testing if a string is square-free. We present an optimal $O(\log\log n)$ time algorithm for testing is a string is square-free, improving the previous bound of $O(\log n)$ given by Apostolico [1] and Crochemore and Rytter [12].

We show matching lower bounds for optimal parallel algorithms that solve the problems above on a general alphabet. The lower bounds for testing if a string is square-free and finding all initial palindromes are derived by a modification of a lower bound for finding the period of a string [7].

*Partially supported by NSF Grant CCR-89-00305, by NIH Library of Medicine Grant R01-LM05118, by AFOSR Grant 90-0107, by NATO Grant CRG 900293 and by the National Research Council of Italy.

†Partially supported by an IBM Graduate Fellowship. Part of the work done while visiting at Università de L'Aquila, L'Aquila, Italy.

‡Partially supported by NSF Grant CCR-90-14605.

1 Introduction

We present optimal CRCW-PRAM algorithms for the problems of finding all periods of a string and testing if a string is square-free. Both solutions are the fastest possible optimal parallel algorithms for these problems over a general alphabet. The two algorithms start with many independent calls to a string matching routine which are performed in parallel and the results of the string matching problems are later combined to give an answer to the problem being solved.

A parallel algorithm is said to be *optimal* if the time-processor product, that is the total number of operations performed, is equal to that of the fastest sequential algorithm. Note that a simple algorithm can compute all periods of a string in constant time if n^2 processors are available. Another simple algorithm can test if a string is square-free using n^3 processors.

A lower bound of $\Omega(\frac{\log n}{\log\log n})$ by Beame and Hastad [4], for computing the parity of n input bits on a CRCW-PRAM with any polynomial number of processors, implies that most interesting problems would take at least that time. However, many problems on strings, including the problems solved in this paper, have trivial CRCW-PRAM algorithms that work in constant time using a polynomial number of processors. This fact suggests that an optimal parallel algorithm that is faster than that lower bound is possible. Our goal is to design fast optimal parallel algorithms.

1.1 Periods

A string $S[1..n]$ has *a period* p if $S[i] = S[i+p]$ for $i = 1 \cdots n-p$. *The period* of S is defined as its shortest period. The period of a string is computed in linear time as a step in Knuth, Morris and Pratt's sequential string matching algorithm [15] and in optimal $O(\log\log n)$ parallel time on a CRCW-PRAM as a step in Breslauer and Galil's string matching algorithm [6]. A recent lower bound by Breslauer and Galil [7] shows that the $O(\log\log n)$ bound is the best possible over a general alphabet, where only comparisons between symbols are allowed. However, Breslauer and Galil's [6] algorithm as well as an algorithm discovered by Vishkin [21] compute the period p only if $p \leq \lceil \frac{n}{2} \rceil$; knowing the fact that $p > \lceil \frac{n}{2} \rceil$ is sufficient to obtain good string matching algorithms. We show that given an optimal parallel algorithm for string matching one can compute all the periods, including those which are longer than half of the length of the input string, in the same processor and time bounds of the string matching algorithm. In particular Breslauer and Galil's [6] algorithm can be used to obtain an optimal $O(\log\log n)$ time CRCW-PRAM algorithm that computes the period of a string.

A *palindrome* is a string which reads the same forward and backward. Formally, $S[1..k]$ is a palindrome if $S[i] = S[k+1-i]$ for $i = 1..k$. A string $S[1..n]$ has an initial palindrome of length k if the prefix $S[1..k]$ is a palindrome. We show how our algorithm can be used to detect all initial palindromes of a string in the same time bound. We can prove also that this is the best time bound possible for any optimal parallel algorithm that solves this problem over a general alphabet. The lower bound is obtained by a modification of a lower bound for string matching of Breslauer and Galil [7] and will be described in the full paper [8].

1.2 Squares

A nonempty string of the form xx is called a *square*. A string that does not contain any square is called *square-free*. For example, the strings *aa*, *abab* and *baba* are squares which are contained in the string *baababa*. It is trivial to show that any string of length larger than three on an alphabet of two symbols contains a square. However, there exist strings of infinite length on a three letter alphabet that are square-free as shown by Axel Thue [19, 20] at the beginning of the century. We develop an efficient parallel algorithm that tests if a string is square-free, improving to $O(\log\log n)$ the previous bound of $O(\log n)$, given by Apostolico [1] and Crochemore and Rytter [12]. A version of our algorithm which will be described in the full paper [2] can detect all squares in the same bounds. We prove also that this is the best time bound possible for an optimal parallel algorithm that solves this problem over a general alphabet.

There exist few sequential algorithms to solve this problem. Algorithms by Apostolico and Preparata [3], by Crochemore [9] and by Main and Lorentz [17] find all the squares in a string of length n in $O(n\log n)$ time. Main and Lorentz [17] also show that $O(n\log n)$ comparisons are necessary even to decide if a string is square-free. In another paper, Main and Lorentz [18] show that the latter problem of deciding whether a string is square-free can be solved in $O(n)$ time if the alphabet is finite. Crochemore [10] also gave a linear time algorithm for this problem.

In parallel, an algorithm by Crochemore and Rytter [12] can test if a string is square-free in optimal $O(\log n)$ time. This algorithm uses $O(n^2)$ space. Other algorithms by Apostolico [1] can test if a string is square-free and even detect all the squares in the same time and processor bounds using only linear auxiliary space. The algorithm for testing if a string is square-free is even more efficient in the case of a finite alphabet and achieves the same time bound of $O(\log n)$ using only $\frac{n}{\log n}$ processors. Apostolico's algorithms [1] assume that the alphabet is ordered, an assumption which is not necessary to solve this problem. All these algorithms are designed for the CRCW-PRAM computation model.

All the parallel algorithms mentioned above are optimal since the time-processor product is $O(n\log n)$ which is the best possible in the case of a general alphabet. Apostolico's [1] algorithm for testing square-freeness in case of finite alphabet is also optimal since the time-processor product is $O(n)$, the best running time of a sequential algorithm for this problem.

The algorithm described in this paper is a parallel version of the sequential algorithm of Main and Lorentz [18].

1.3 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. In case that several processors attempt to write simultaneously at the same memory location, we assume they always

attempt to write the same value.

Our algorithms use a string matching algorithm as a "black-box" to find all occurrences of a short string in a longer string. The input to the string matching algorithm will consist 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 the Breslauer and Galil [6] parallel string matching algorithm that takes $O(\log\log n)$ time on a $\frac{n}{\log\log n}$-processor CRCW-PRAM. This algorithm is the fastest optimal parallel string matching algorithm on a general alphabet as implied by a lower bound of Breslauer and Galil [7]. If a faster string matching algorithm on a finite alphabet exists, it would imply a faster algorithm for finding the periods and for testing if a string is square-free.

We use also an algorithm of Fich, Ragde and Wigderson [13] to compute the minimum of n integers in the between 1 and n in constant time using an n-processor CRCW-PRAM. We use this algorithm, for example, to find the first occurrence of a string in an other string. After the occurrences are computed by the string matching algorithm mentioned above, we look for the smallest i such that $match[i] = true$.

Finally, we use the following theorem:

Theorem 1.1 (Brent): Any synchronous 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.

This theorem can be used for example to slow down a constant time p-processor algorithm to work in time t using p/t processors. Coming 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 on n processors, while the call to the string matching procedure takes $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors. By Theorem 1.1 the second step can be slowed down to work in $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors.

2 Finding the periods

We describe an algorithm that given a string $S[0..n]$ will compute all the periods of S. The output of the algorithm will be a Boolean array $P[1..n]$ such that $P[i] = true$ if and only if i is a period of S. Note that, for the convenience of the presentation, in this section the input string $S[0..n]$ is of length $n+1$ and starts with $S[0]$.

We will prove the following theorem:

Theorem 2.1: There exists an algorithm to compute $P[1..n]$ that takes $O(\log\log n)$ time using $\frac{n}{\log\log n}$ processors.

Corollary 2.2: The period of a string S can be computed in the same time and processor bounds.

Proof: The period of S is the smallest i such that $P[i]$ is *true*. We use the technique of Fich, Ragde and Wigderson [13] to compute the minimum of n integers in the range $1..n$ in constant time using an n-processor CRCW-PRAM. (This step can be slowed down to work in optimal $O(\log\log n)$ time by Theorem 1.1.) □

Corollary 2.3: All initial palindromes of a string S can be computed in the same time and processor bounds.

Proof: Suppose we want to compute all initial palindromes of a string w that does not contain the symbol \$. We present $w\$w^R$ (where w^R is the string w reversed) as input to the algorithm that computes all periods of a string. Each period of this string corresponds to an initial palindrome of w. Two copies of the string $w\$w^R$ are aligned with each other shifted by some offset and the overlapping parts are identical if and only if the overlapping part is an initial palindrome of w. This reduction was used by Fischer and Paterson [14]. □

Example: The string *abaab* has an initial palindrome *aba*. This initial palindrome corresponds to the period *abaab\$ba* of the string *abaab\$baaba*.

Proof of Theorem 2.1:

The algorithm will proceed in independent stages which are all computed simultaneously and described in the next section. In stage number η, $0 \leq \eta < m$, we will compute only $P[n - l_\eta + 1..n - l_{\eta+1}]$; where the sequence $\{l_\eta\}$ is a decreasing sequence defined as $l_0 = n$, $l_{\eta+1} = \lfloor \frac{2}{3} l_\eta \rfloor$ and m is the smallest integer for which $l_m = 0$. Note that each stage is assigned to compute a disjoint part of the output array P and the entire array is covered.

We denote by T_η the time it takes to compute stage number η using P_η processors. The number of operations at stage η will be denoted by $O_\eta = T_\eta P_\eta$. We show later how to implement stage number η in $T_\eta = O(\log\log l_\eta)$ time and $O_\eta = l_\eta$ operations using Breslauer and Galil's [6] parallel string matching algorithm.

Since all stages of our algorithm are executed in parallel the total number of operation performed in all stages is $\sum_\eta O_\eta \leq \sum_\eta \left(\frac{2}{3}\right)^\eta n = O(n)$ and the time is $\max T_\eta = O(\log\log n)$. By Theorem 1.1 the algorithm can be implemented using $\frac{n}{\log\log n}$ processors in $O(\log\log n)$ time. □

2.1 A single stage

We describe a single stage η, $0 \leq \eta < m$, that computes $P[n - l_\eta + 1..n - l_{\eta+1}]$ in optimal $O(\log\log l_\eta)$ time. Note, that since a period p implies that $S[0..n-p] = S[p..n]$, there must be an occurrence of $S[0..l_{\eta+1}]$ starting at each position p which is a period of S in the range computed by this stage.

We start with a call to a string matching algorithm to find all occurrences of $S[0..l_{\eta+1}]$ in $S[n - l_\eta + 1..n]$. Let q_i, $i = 1..r$, denote the indices of all these occurrences (all indices are in the string $S[0..n]$, thus $n - l_\eta < q_i \leq n - l_{\eta+1}$).

If there were no occurrences found, the string S has no period in the range computed by this stage and all entries of $P[n - l_\eta + 1..n - l_{\eta+1}]$ can be set to $false$. Otherwise, we continue with another call to a string matching algorithm to find all occurrences of $S[0..l_{\eta+1}]$ in $S[0..l_\eta - 1]$. Let p_i, $i = 1..k$, denote the indices of all these occurrences (note that $p_1 = 0$).

If there was only one occurrence of $S[0..l_{\eta+1}]$ in $S[n - l_\eta + 1..n]$, it can be verified to be a period in $O(l_\eta)$ operations. However, if there are $r > 1$ occurrences, $O(r l_\eta)$ operations may be needed to verify all of them. Luckily the sequences $\{p_i\}$ and $\{q_i\}$ have a "nice" structure

as we show in the following lemmas. This structure enables us to proceed efficiently to test which of the q_i's is actually a period of S.

Lemma 2.4 (Lyndon and Schutzenberger [16]): If a string of length m has two periods of length p and q and $p+q \leq m$, then it has also a period of length $\gcd(p,q)$.

Lemma 2.5: If a string $A[1..l]$ has period p and occurs only at positions $p_1 < p_2 < \cdots < p_k$ of a string $B[1..\lceil \frac{3}{2}l \rceil]$, then the p_i's form an arithmetic progression with difference p.

Proof: Assume $k \geq 2$. We prove that $p = p_{i+1} - p_i$ for $i = 1 \cdots k-1$. The string $A[1..l]$ has periods p and $q = p_{i+1} - p_i$. Since $p \leq q \leq \lceil \frac{l}{2} \rceil$, by Lemma 2.4 it has also a period of length $gcd(p,q)$. But p is the shortest period so $p = gcd(p,q)$ and p must divide q. The string $B[p_i..p_{i+1}+l-1]$ has period p. If $q > p$ then there must be another occurrence of A at position $p_i + p$ of B, a contradiction. □

Lemma 2.6: The sequences $\{p_i\}$ and $\{q_i\}$ form an arithmetic progression with difference $\mathcal{P}$, where $\mathcal{P}$ is the period of $S[0..l_{\eta+1}]$.

Proof: The sequences p_i and q_i are indices of occurrences of a string of length $l_{\eta+1}+1$ in strings of length l_η. Recall that $l_{\eta+1} = \lfloor \frac{2}{3} l_\eta \rfloor$. By Lemma 2.5 the p_i's and q_i's form an arithmetic progression with a difference $\mathcal{P}$, the period of $S[0..l_{\eta+1}]$. □

The sequences $\{p_i\}$ and $\{q_i\}$ can be represented using three integers (each): the start of the sequence, the difference, and the length of each sequence. This representation can be easily obtained from the output of the string matching algorithm in constant time and l_η processors.

Some of the q_i's can be ruled out of being periods of S immediately as we show in the following lemma.

Lemma 2.7: If $k < r$ then q_i is not a period of S for $1 \leq i \leq r-k$.

Proof: Assume q_i is a period of S and $1 \leq i \leq r-k$. In this case $S[q_i..n] = S[0..n-q_i]$. The string $S[q_i..n]$ has $r-i+1 > k$ occurrences of $S[0..l_{\eta+1}]$, which are $q_i \cdots q_r$. But $S[0..n-q_i]$ has only k occurrences of $S[0..l_{\eta+1}]$; contradiction. □

There might be two reasons why $q_r + \mathcal{P}$ is not included in the $\{q_i\}$ sequence:

1. If $S[q_r + \mathcal{P}..\mathcal{N}] \neq S[0..\mathcal{N} - q_r - \mathcal{P}]$, and $\mathcal{N} = \min(n, q_r + \mathcal{P} + l_{\eta+1})$ we call it a **mismatch**.

2. If there is no mismatch then the only reason that $q_r + \mathcal{P}$ is not in the $\{q_i\}$ sequence is that $q_r + \mathcal{P} + l_{\eta+1} > n$. We call this case **an overflow**.

Lemma 2.8 (a mismatch): If $S[q_r + \mathcal{P}..\mathcal{N}] \neq S[0..\mathcal{N} - q_r - \mathcal{P}]$ then, S has at most one period in the range computed by this stage. This only possible period may exist if $k \leq r$ and it is q_{r-k+1}.

Proof: By Lemma 2.7 all q_i, $1 \leq i < r-k+1$, are not periods. Assume q_i is a period and $i > r-k+1$, then $S[q_i..n] = S[0..n-q_i]$. Since $r-i+2 \leq k$ and $p_j = (j-1)\mathcal{P}$, also $S[q_r+\mathcal{P}..\mathcal{N}] = S[p_{r-i+2}..\mathcal{N}-q_i]$. By the assumption of a mismatch $S[q_r+\mathcal{P}..\mathcal{N}] \neq S[0..\mathcal{N}-q_r-\mathcal{P}]$. So $S[p_{r-i+2}..\mathcal{N}-q_i] \neq S[0..\mathcal{N}-q_r-\mathcal{P}]$. But $S[p_{r-i+2}..p_{r-i+2}+l_{\eta+1}] = S[0..l_{\eta+1}]$ and also $\mathcal{N} - q_r - \mathcal{P} \leq l_{\eta+1}$; contradiction. □

Lemma 2.9 (an overflow): If $S[q_r + \mathcal{P}..n] = S[0..n - q_r - \mathcal{P}]$ then:

a. If $r < k$ then $q_1, .., q_r$ are periods of S.

b. If $r \geq k$ then $q_{r-k+2}, .., q_r$ are periods of S. In this case q_{r-k+1} can also be a period of S.

Proof: Assume $S[q_r + \mathcal{P}..n] = S[0..n - q_r - \mathcal{P}]$. It is enough to show that q_i is a period of S, for $\max(r - k + 2, 1) \leq i \leq r$.

By the definitions of the $\{q_i\}$ and $\{p_i\}$ sequences

$$S[0..p_{r-i+1} + l_{\eta+1}] = S[q_i..q_r + l_{\eta+1}] \tag{1}$$

since both substrings are covered by $r - i + 1$ occurrences of $S[0..l_{\eta+1}]$. Since $r - i + 2 \leq k$ also

$$S[p_{r-i+2}..p_{r-i+2} + l_{\eta+1}] = S[0..l_{\eta+1}]. \tag{2}$$

But $n - q_r - \mathcal{P} < l_{\eta+1}$ and $S[q_r + \mathcal{P}..n] = S[0..n - q_r - \mathcal{P}]$. By taking prefixes of (2)

$$S[q_r + \mathcal{P}..n] = S[p_{r-i+2}..p_{r-i+2} + n - q_r - \mathcal{P}]. \tag{3}$$

By combining equalities (1) and (3), we get that $S[0..n - q_i] = S[q_i..n]$. □

The computation in stage η can be summarized as follows:

1. Compute the $\{q_i\}$ and $\{p_i\}$ sequences.
2. If $k \leq r$, check if q_{r-k+1} is a period of S.
3. If $S[q_r + \mathcal{P}..\mathcal{N}] = S[0..\mathcal{N} - q_r - \mathcal{P}]$ then,
   a. If $r < k$, then $q_1, .., q_r$ are all periods of S.
   b. If $r \geq k$, then $q_{r-k+2}, .., q_r$ are all periods of S.

Lemma 2.10: Stage number η is correct and it takes $O(\log\log l_\eta)$ time and $O(l_\eta)$ operations.
Proof: Correctness of the algorithm follows from Lemmas 2.7, 2.8 and 2.9. The two calls to a string matching algorithm to compute the $\{q_i\}$ and $\{p_i\}$ sequences take $O(\log\log l_\eta)$ time and $O(l_\eta)$ operations if we use Breslauer and Galil's [6] string matching algorithm. The sequences $\{q_i\}$ and $\{p_i\}$ can be represented by three integers which can be computed from the output of the string matching algorithm (which is assumed to be a Boolean vector representing all occurrences) in constant time and $O(l_\eta)$ operations. Steps *2* and *3* can be done also in constant time and $O(l_\eta)$ operations. □

2.2 A lower bound

The algorithm that finds all periods of a string described above achieves the best time bound possible for an optimal parallel algorithm as implied by a lower bounds of Breslauer and Galil [7]. This lower bound can be modified to a lower bound for finding the initial palindromes of a string.

The reduction from periods to inital palindromes of Corollary 2.3 does not imply the immediate translation of the lower bound to this problem. The details of the lower bound will be given in the full paper [8].

3 Looking for squares

We describe an algorithm that tests if a string $S[1..n]$ is square-free in $O(\log\log n)$ time on a $\frac{n\log n}{\log\log n}$-processor CRCW-PRAM. This algorithm is optimal since the time-processor product is $O(n\log n)$, the same as the running time of the fastest sequential algorithm on general alphabet.

We will prove the following theorem:

Theorem 3.1: There exists an algorithm that tests if a string $S[1..n]$ is square-free and takes $O(\log\log n)$ time using $\frac{n\log n}{\log\log n}$ processors.

The algorithm will work in independent stages which are all computed simultaneously. In stage number η, $1 \leq \eta \leq \lceil \log_2 n - 1 \rceil$, we look only for squares xx where $2^\eta - 1 \leq |x| < 2^{\eta+1} - 1$. If a square is found, a global variable will be set to indicate that the string is not square-free. Note that the complete range of possible lengths of x is covered and if there exist a square it will be discovered.

We denote by T_η the time it takes to compute stage number η on P_η processors. The number of operations performed in stage η will be denoted by $O_\eta = T_\eta P_\eta$.

We show later how to implement stage η in $T_\eta = O(\log\log 2^\eta)$ time and $O_\eta = O(n)$ operations using Breslauer and Galil's [6] string matching algorithm. Since there are $O(\log n)$ stages, the total number of operations is $O(n\log n)$. By Theorem 1.1 the complete algorithm can be implemented in $\max T_\eta = O(\log\log n)$ time and $\frac{n\log n}{\log\log n}$ processors.

3.1 A single stage

We describe a single stage η, $1 \leq \eta \leq \lceil \log_2 n - 1 \rceil$, that looks only for squares xx where $2^\eta - 1 \leq |x| < 2^{\eta+1} - 1$. Every comparison to a symbol $S[q]$ out of the range $S[1..n]$ is assumed to be answered as unequal.

Partition the input string $S[1..n]$ into blocks of length l_η, where l_η is defined as $l_\eta = 2^{\eta-1}$. That is, block number k, for $1 \leq k < \frac{n}{l_\eta}$, is $S[(k-1)l_\eta + 1..kl_\eta]$. Stage η consists of sub-stages which are also computed simultaneously. There is a sub-stage for each block of length l_η, in which the algorithms checks if there is a square xx such that $2^\eta - 1 \leq |x| < 2^{\eta+1} - 1$ and the first x contains that block.

Each such sub-stage starts with a call to the string matching algorithm to find all occurrences of the k^{th} block, $S[(k-1)l_\eta+1..kl_\eta]$, in $S[(k+1)l_\eta..(k+4)l_\eta-2]$. Let $p_1 < p_2 < \cdots < p_r$, be the indices of these occurrences. Then $(k+1)l_\eta \leq p_i \leq (k+3)l_\eta - 1$ for $1 \leq i \leq r$.

Note, that for each square xx such that $2^\eta - 1 \leq |x| < 2^{\eta+1} - 1$ and the first copy of x contains the block $S[(k-1)l_\eta + 1..kl_\eta]$ there must be an occurrence of $S[(k-1)l_\eta + 1..kl_\eta]$ at position $(k-1)l_\eta + |x| + 1$. This occurrence is included in the $\{p_i\}$ sequence.

Lemma 3.2: For each p_i, we can verify in constant time and $O(l_\eta)$ operations if there is a square xx of length $p_i - (k-1)l_\eta - 1$ that contains the block $S[(k-1)l_\eta + 1..kl_\eta]$.

Proof: Let $l = p_i - (k-1)l_\eta - 1$ be the length of the square we are looking for. For all ζ in the range $kl_\eta < \zeta \leq (k-1)l_\eta + l$ check if $S[\zeta - l] = S[\zeta]$ and if $S[\zeta] = S[\zeta + l]$. Let ζ_L be the largest index in this range such that $S[kl_\eta + 1..\zeta_L] = S[kl_\eta + l + 1..\zeta_L + l]$ and ζ_R be the smallest index such that $S[\zeta_R..(k-1)l_\eta + l] = S[\zeta_R - l..(k-1)l_\eta]$. Using the algorithm of Fich, Ragde and Wigderson [13] we can find ζ_L and ζ_R in constant time and $O(l_\eta)$ operations.

We show that there is a substring xx of length l containing the block $S[(k-1)l_\eta + 1..kl_\eta]$ if an only if $\zeta_L \geq \zeta_R - 1$. Recall that since there is an occurrence of $S[(k-1)l_\eta + 1..kl_\eta]$ at position p_i, we know that $S[(k-1)l_\eta + 1..kl_\eta] = S[p_i..p_i + l_\eta - 1]$.

If $\zeta_L \geq \zeta_R - 1$ then $S[\zeta_R - l..\zeta_L] = S[\zeta_R..\zeta_L + l]$. The last equality means that there are squares of length l starting at positions $\zeta_R - l$ up to $\zeta_L - l + 1$.

On the other hand, if there is a square xx such that the first x contains $S[(k-1)l_\eta+1..kl_\eta]$ that starts at position π, then for all ζ in the range $\pi+|x| \leq \zeta \leq (k-1)l_\eta+l$, $S[\zeta-|x|] = S[\zeta]$ and for all ζ in the range $kl_\eta < \zeta < \pi + |x|$, $S[\zeta] = S[\zeta + |x|]$. But $\zeta_L < \zeta_R - 1$ so $S[\zeta_L + 1] \neq S[\zeta_L + |x| + 1]$ and $S[\zeta_R - |x| - 1] \neq S[\zeta_R - 1]$; a contradiction to the existence of a square of length $|x|$ at position π. □

We could verify all the p_i's in constant time using Lemma 3.2 but it might take $O(rl_\eta)$ operations if the length of the $\{p_i\}$ sequence is r. Luckily, we do not have to verify all the p_i's. If the length of the $\{p_i\}$ sequences $r > 2$ then there is a square as the following lemma shows:

Lemma 3.3: If the number of occurrences of $S[(k-1)l_\eta+1..kl_\eta]$ in $S[(k+1)l_\eta..(k+4)l_\eta-2]$ is larger than two then S[1..n] has a square. This square is actually shorter than the squares which are supposed to be found by this stage.

Proof: All the occurrences $\{p_i\}$, $1 \leq i \leq r$ satisfy $(k+1)l_\eta \leq p_i \leq (k+3)l_\eta - 1$. If $r \geq 3$ then either $p_2 - p_1 \leq l_\eta$ or $p_3 - p_2 \leq l_\eta$. In this case there is a square of length $p_2 - p_1$ or $p_3 - p_2$ (respectively) starting at position p_1 or p_2 (respectively). □

The computation in each sub-stage of stage η can be summerized 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 $S[1..n]$ has a square. This square will be found also by some stage μ, $\mu < \eta$.
3. If the $\{p_i\}$ sequence has at most two elements, check if these elements correspond to squares using the procedure described in Lemma 3.2.

Note that if the $\{p_i\}$ sequence has only a small number of elements we can actually find all the squares efficiently using Lemma 3.2. The complete details of an algorithm that finds all the squares will be described in the full paper [2].

Lemma 3.4: Stage number η is correct and it takes $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors.

Proof: For correctness we have to show that if the string $S[1..n]$ has a any square xx, $2^\eta - 1 \leq |x| < 2^{\eta+1} - 1$, then some square will be found.

Assume there is such a square. 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 assigned to that block will either find the square xx or conclude that there is a shorter square by Lemma 3.3. In both cases a square has been found. Note that some squares can be detected by several sub-stages simultaneously.

Stage η consists of $\frac{n}{l_\eta}$ independent sub-stages. In each sub-stage, step number 1 takes $O(\log\log l_\eta)$ time and $O(l_\eta)$ operation. Steps number 2 and 3 take constant time and $O(l_\eta)$ operations. Since all sub-stages are computed in parallel the time stage η takes is $O(\log\log l_\eta)$ and the total number of operations performed is $\frac{n}{l_\eta} l_\eta = O(n)$. □

Theorem 3.1 follows from the last lemma since the $\log n$ stages are executed in parallel.

3.2 A Lower Bound

We prove a lower bound for testing if a string is square-free by a reduction to the lower bound for string matching by Breslauer and Galil [7]. This lower bound is on the number of comparison rounds the algorithm performs when there are p comparisons in each round. This bound holds for the CRCW-PRAM model in case of a general alphabet where the only access an algorithm has to the input strings is by comparison of symbols.

Breslauer and Galil [7] show that an adversary can fool any algorithm which claims to check if a string has a period shorter than half of its length in less than $\Omega(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ rounds of p comparisons each.

We will not go into the complete details of that lower bound. We only use the fact that the adversary of Breslauer and Galil [7] answers the comparisons in each round in such a way that after $\Omega(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ rounds it is still possible that the string $S[1..n]$ has a period shorter than half of its length, or that it does not have any such period. The string generated by that adversary has also the following property: If $S[i] = S[j]$, for $i < j$, then for any integer k, such that $k \equiv i \, mod \, (j-i)$ and $1 \leq k \leq n$, $S[k] = S[i]$.

Lemma 3.5: The string generated by the adversary of Breslauer and Galil has a period smaller than half of its length if and only if it has a square.

Proof: If the string generated by the adversary has a period which is shorter than half the length of the string then there is a square starting at the beginning of the string, which is this period repeated twice.

It remains to show that if there is a square then the string has a period shorter than half of its length. Assume that a square of length $2l$, starts at position i. This means that $S[i+k] = S[i+l+k]$ for $k = 0..l-1$. By the property of the string generated by the

adversary, the string $S[1..n]$ has a period l, which is smaller than half the length of S. □

Now, we are ready to prove the lower bound.

Theorem 3.6: Any optimal parallel algorithm that tests if a string $S[1..n]$ is square-free takes $\Omega(\log\log n)$ time.

Proof: Any optimal algorithm performs at most $n \log n$ comparisons in each round since the fastest sequential algorithm that solves this problem takes $O(n \log n)$ time as shown by Main and Lorentz [17]. Assume an algorithm performs $p = n \log n$ comparisons in each round. By the lower bound of Breslauer and Galil [7] and Lemma 3.5 after $\Omega(\log\log n)$ rounds the adversary can decide if $S[1..n]$ is square-free or not, fooling any algorithm which terminates in less rounds. □

Optimal $O(\log n)$ time algorithms for detecting squares designed by Apostolico [1] work under the assumption that the alphabet is ordered. The following corollary shows that alphabet order can not help.

Corollary 3.7: The same lower bound holds even if order comparisons, which result in *less than*, *equal* or *greater than* answers are allowed instead of the *equal* or *unequal* type of comparisons.

Proof: Breslauer and Galil's [7] lower bound holds also in this case. □

References

[1] Apostolico, A. (1991), Optimal parallel detection of squares in strings, *CS-TR-91-026, Purdue and Algorithmica, in press.*

[2] Apostolico, A. and Breslauer D. (1991), An optimal $O(\log\log n)$ time parallel algorithm for detecting all repetitions in a string, *In preparation.*

[3] Apostolico, A. and Preparata, F. P. (1983), Optimal off-line detection of repetitions in a string, *Theoretical Computer Science 22*, 297-315.

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

[5] Brent, R. P. (1974), The parallel evaluation of general arithmetic expressions, *J. ACM 21*, 201-206.

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

[7] Breslauer, D. and Galil, Z. (1991), A lower bound for parallel string matching, *Proc. 23nd ACM Symp. on Theory of Computation*, 439-443.

[8] Breslauer, D. and Galil Z. (1991), Finding all the periods and initial palindromes of a string in parallel, *manuscript.*

[9] Crochemore, M. (1981), An optimal algorithm for computing the repetitions in a word, *Information Processing Letters 12:5*, 244-250.

[10] Crochemore, M. (1986), Transducer and repetitions, *Theoretical Computer Science 45*, 63-86.

[11] Crochemore, M. and Rytter, W. (1990), Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays, manuscript.

[12] Crochemore, M. and Rytter, W. (1991), Efficient Parallel Algorithms to Test Squarefreeness and Factorize Strings, *Information Processing Letters 38*, 57-60.

[13] Fich, F. E., Ragde, R. L., and Wigderson, A. (1984), Relations between concurrent-write models of parallel computation, *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, 179-189.

[14] Fischer, M. J. and Paterson, M. S. (1974), String-Matching and other products, *SIAM-AMS proceedings, Vol 7*, 113-125.

[15] Knuth, D. E., Morris, J. H. and Pratt, V. R. (1977), Fast pattern matching in strings, *SIAM J. Comput. 6*, 322-350.

[16] Lyndon, R. C. and Schutzenberger, M. P. (1962), The equation $a^M = b^N c^P$ in a free group, *Michigan Math. J. 9*, 289-298.

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

[18] Main, G. M. and Lorentz, R. J. (1985), Linear time recognition of squarefree strings, in *Combinatorial Algorithms on Words*, Edited by A. Apostolico and Z. Galil, 271-278.

[19] Thue, A. (1906), Über unendliche Zeichenreihen, *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, Nr. 7, 1-22.

[20] Thue, A. (1912), Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen, *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, Nr. 1, 1-67.

[21] Vishkin, U. (1985), Optimal parallel pattern matching in strings, *Information and Control 67*, 91-113.