BRICS RS-94-42 Breslauer & Gąsieniec: Efficient String Matching on Coded Texts

Efficient String Matching on Coded Texts

Dany Breslauer
Leszek Gąsieniec

BRICS Report Series RS-94-42

ISSN 0909-0878 December 1994

Efficient String Matching on Coded Texts

Dany Breslauer[*] Leszek Gąsieniec[†]

Abstract

The so called "four Russians technique" is often used to speed up algorithms by encoding several data items in a single memory cell. Given a sequence of n symbols over a constant size alphabet, one can encode the sequence into $O(n/\lambda)$ memory cells in $O(\log\lambda)$ time using $n/\log\lambda$ processors.

This paper presents an efficient CRCW-PRAM string-matching algorithm for coded texts that takes $O(\log\log(m/\lambda))$ time[1] making only $O(n/\lambda)$ operations, an improvement by a factor of $\lambda = O(\log n)$ on the number of operations used in previous algorithms. Using this string-matching algorithm one can test if a string is square-free and find all palindromes in a string in $O(\log\log n)$ time using $n/\log\log n$ processors.

1 Introduction

In the string-matching problem one is searching for occurrences of a pattern string $\mathcal{P}[1..m]$ in a text string $\mathcal{T}[1..n]$. There exist several $O(n+m)$ time sequential string-matching algorithms that are used in a large variety of applications. Galil [23] published the first efficient parallel string-matching algorithm. His algorithm takes $O(\log m)$ time and uses n processors in the concurrent-read concurrent-write parallel random-access-machine model. If the symbols of the input strings are taken from a constant size alphabet, then the number of processors is reduced to $n/\log m$, achieving an *optimal speedup*, or in other words achieving a time-processor product that is equal to the running time of the fastest sequential algorithm for the problem. (Notice that there is a trivial constant time parallel string-matching algorithm that uses nm processors. Our goal is to design fast parallel algorithms that use few processors.) The saving is obtained by using the so called "four Russians technique", named after the work of Arlazarov et al. [8], where each block of $O(\log m)$ symbols is packed into a single memory cell to facilitate comparisons of many symbols in a single operation.

[*]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). Part of the research reported in the paper was carried out while this author was 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.

[†]Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warszawa, Poland.

[1]Throughout the paper $\log n$ usually means $\max(1, \log_2 n)$.

Vishkin [38] generalized Galil's algorithm and obtained an $O(\log m)$ time algorithm that uses only $n/\log m$ processors, regardless of the alphabet size. Breslauer and Galil [10] gave an $O(\log\log m)$ time string-matching algorithm that uses $n/\log\log m$ processors. Breslauer and Galil [11] proved that if $n = O(m)$, then this is the best time bound achievable by an optimal-speedup string-matching algorithm that has access to the input strings only by pairwise symbol comparisons.

Vishkin [39] presented an optimal-speedup string-matching algorithm that takes $O(\log^2 m)$ time for the pattern preprocessing and then only $O(\log^* m)$ time to find all occurrences of the pattern in the text. Galil [24] improved the text processing step to constant time. Goldberg and Zwick [26] presented an algorithm with a tradeoff between the the time spent in the pattern preprocessing and the text processing steps. Recently, Crochemore et al. [16] discovered an algorithm that takes $O(\log\log m)$ time to preprocess the pattern and then constant time to find all occurrences of the pattern in the text. Crochemore et al. also gave a randomized version of their pattern preprocessing algorithm that takes only constant expected time. These algorithms access the input strings by pairwise symbol comparisons and do not require any special assumption on the alphabet size.

This paper gives a variant of Breslauer and Galil's [10] string-matching algorithm that takes $O(\log\log(m/\lambda))$ time making only $O(n/\lambda)$ operations, after the input strings are coded in $O(n/\lambda)$ memory cells. The parameter $\lambda = O(\log n)$. The input symbols, which are assumed to be taken from a constant size alphabet, are encoded in $O(\log\lambda)$ time using $n/\log\lambda$ processors. Notice that the encoding step dominates the number of operations made. Thus the new algorithm is inferior to the previously known parallel string-matching algorithms since it has the additional restriction on the alphabet size. However, the advantages of the algorithm become clear if the input strings are given in their coded form.

Apostolico, Breslauer and Galil gave efficient parallel algorithms for testing if a string is square-free and for finding all palindromes in a string [4, 5, 6]. Their algorithms share a similar structure, take $O(\log\log n)$ time utilizing $n\log n/\log\log n$ processors, and rely on a procedure that is used to solve several string-matching problems. Observing that it suffices to encode the input string only once and use the coded string as input to many string-matching problem instances, we improve the processor bounds of these algorithms and obtain optimal-speedup $O(\log\log n)$ time $n/\log\log n$-processor algorithms for the two problems. We assume that the reader is familiar with these algorithms and with the Breslauer-Galil string-matching algorithm.

The paper is organized as follows. Section 2 introduces the computation model. Section 3 describes how the input strings are encoded and how the coded strings are manipulated. The string-matching algorithm is given is Section 4 and its applications for testing if a string is square-free and for finding all palindromes in a string are given in Section 5. Concluding remarks and open problems are given in Section 6.

2 The computation model

The computation model we use in this paper is the *common concurrent-read concurrent-write parallel random-access-machine.* In this model, processors are allowed to read and write simultaneously at the same memory location. If many processors write to the same memory cell at the same time they are guaranteed to write the same value. The arithmetic operations $+$, $-$, $\times$, and integer division $/$ can be performed by each processor in constant time on any memory words. Notice that the memory words must be able to hold numbers which are as large as the lengths of the input strings.

The following lemma is often used in parallel algorithms. The claimed bounds hold also in the weaker exclusive-read exclusive-write parallel random-access-machine model.

Lemma 2.1 *(Lander and Fischer [29]) Given a sequence $x_1, \ldots, x_h$, and an associative binary operation $\oplus$, one can compute the prefix sums $x_1 \oplus x_2 \oplus \cdots \oplus x_g$, for all $g = 1, \ldots, h$, in $O(\log h)$ time using $h/\log h$ processors.*

In the CRCW-PRAM model, certain computations can be carried out much faster.

Lemma 2.2 *(Fich, Ragde and Wigderson [20]) Given a collection of h integers from the range $1, \ldots, h$, it is possible to find their minima value in constant time using an h-processor CRCW-PRAM.*

The last lemma will be used mainly to find the leftmost non-zero entry in an array. We shall also use the following general theorem without going into the details of the assignment of processors to their tasks.

Theorem 2.3 *(Brent [9]) Any parallel algorithm of time t that consists of a total of x elementary operations can be implemented on p processors in $O(\lceil x/p \rceil + t)$ time.*

3 Encoding strings

Throughout the paper we assume that the input alphabet is $\Sigma = \{0, 1, \ldots, c-1\}$, for some fixed positive constant c. Since the memory words in our model are able to store numbers as large as n, where n is the length of the string $\mathcal{S}[1..n]$ being encoded, we could represent at least $\lfloor \log_c n \rfloor$ symbols in each memory word as a number in base c that has the symbols as its digits.

The new string-matching algorithm takes advantage of the coded representation of strings in two ways: fast comparison of blocks of several symbols and table lookup of precomputed information. While the first use would benefit from packing as many symbols as possible in each memory word, the second might require a substantial use of computational resources (time, processors, space) to compute and store the tables. The balance is achieved by packing only $\lambda = \max(1, \lfloor \frac{1}{8} \log_c n \rfloor)$ symbols in each word. The parameters c and λ will be used throughout the paper.

Given a string $\mathcal{S}[1..n]$, we break the string into consecutive blocks of λ symbols and encode each block into a memory word. Thus, a string of length n is encoded into a sequence of $\lceil n/\lambda \rceil$ memory words. We shall continue to refer to the symbols, the indices and the length of the original string, using the encoded representation only when we wish to compare substrings fast or when we wish to look up some information that we have precomputed for the coded strings.

To manipulate the coded strings efficiently we extend the repertoire of operations supported by our model to include the powers c^h, for $h = 0, \ldots, \lambda$, and to support the modulo operation. The modulo operation can be implemented as $a \bmod b = a - b * \lfloor a/b \rfloor$, and the powers c^h are implemented by a table lookup.

Lemma 3.1 *Given a string $\mathcal{S}[1..n]$ over a constant size alphabet, one can encode the string into $O(n/\lambda)$ memory words in $O(\log \lambda) = O(\log \log n)$ time using $n/\log \lambda = O(n/\log \log n)$ processors.*

Proof: The encoding consists of the string representation as a sequence of base c numbers together with some lookup tables. Most of these tables are described only later at the place where they are used, but their creation takes place when the string $\mathcal{S}[1..n]$ is being encoded and they are considered part of the encoded representation.

The table of powers of c mentioned above is precomputed by Lemma 2.1 in $O(\log \lambda)$ time making $O(\lambda)$ operations. It occupies $O(\lambda)$ space. Notice that the power table and other tables that are described later depend only on the parameters c and λ. The size of each table will not exceed $O(n/\lambda)$ and the time to create each table will not exceed $O(\log \lambda)$ making at most $O(n)$ operations.

The string representation is created by encoding each consecutive block of symbols $\mathcal{S}[g], \ldots, \mathcal{S}[g+\lambda-1]$, as a base c number $\mathcal{S}[g] + \mathcal{S}[g+1] * c + \cdots + \mathcal{S}[g+\lambda-1] * c^{\lambda-1}$. By Lemma 2.1, this computation is done in $O(\log \lambda)$ time making $O(\lambda)$ operations. Since all the $\lceil n/\lambda \rceil$ λ-blocks are encoded simultaneously, the encoding takes $O(\log \lambda)$ time making $O(n)$ operations. By Theorem 2.3, the whole encoding step takes $O(\log \lambda)$ time using $n/\log \lambda$ processors. $\Box$

Using the encoded representation, we can save a factor of λ in the number of operations needed to compare two strings.

Lemma 3.2 *It is possible to compare two coded strings of original length l and to find the position of the first mismatch between them if they are not equal, in constant time and $O(\lceil l/\lambda \rceil)$ operations.*

Proof: The algorithm will use a precomputed table $\mathcal{CMP}[\hat{a}, \hat{b}]$ that gives the position of the first mismatch between the strings $\hat{a}$ and $\hat{b}$. We use the notation $\hat{a}$ and $\hat{b}$ to refer to both the integers that code λ symbols and to the string formed by these symbols. The size of the $\mathcal{CMP}$ table is $O(c^{2\lambda}) \leq O(n/\lambda)$ and it can be computed in constant time making $O(c^{2\lambda} \times \lambda) \leq O(n)$ operations. We describe how the computation of this table is carried out. The computation of the other tables that are mentioned later is similar and will not be described in such detail.

Each entry of the table $\mathcal{CMP}[\hat{a},\hat{b}]$ is computed independently and simultaneously by λ processors. Notice that if symbols are indexed from 1 to λ, then the k^{th} symbol of $\hat{a}$ is given by the formula: $\lfloor \hat{a}/c^{k-1} \rfloor \bmod c$. The symbols of $\hat{a}$ and $\hat{b}$ are extracted from the integer representation of these strings and the corresponding symbols are compared simultaneously. The position of the first mismatch is found by Lemma 2.2 in constant time making $O(\lambda)$ operations, and is assigned to $\mathcal{CMP}[\hat{a},\hat{b}]$.

Observe that the strings being compared might be specified by indices in some longer coded strings. Thus, their coded representations do not necessarily starts on the boundaries of the memory words. Therefore, the algorithm first extracts proper $\lceil l/\lambda \rceil$ words that constitute the coded representation of each of the two strings. Notice that the coded representation of the substring of length λ starting at position $k \geq 2$ of the string coded as $\hat{a}$ followed by $\hat{b}$ is given as: $\lfloor \hat{a}/c^{k-1} \rfloor + c^{\lambda-k} \times (\hat{b} \bmod c^{k-1})$.

The algorithm then compares the extracted coded representations and finds the leftmost coded words where the strings disagree in constant time and $O(\lceil l/\lambda \rceil)$ operations by Lemma 2.2. Then, using the table $\mathcal{CMP}$ it finds the actual symbol within this memory words where the strings disagree. $\square$

4 String matching with coded strings

In this section we describe an algorithm that finds all occurrences of a pattern $\mathcal{P}[1..m]$ in a text $\mathcal{T}[1..n]$. The input strings are assumed to be given in their coded form with the coding parameter λ. The algorithm takes $O(\log\log(m/\lambda))$ time and makes $O(\lceil n/\lambda \rceil)$ operations. If the strings are not already coded, one can encode them as the single string $\mathcal{S}[1..n+m] = \mathcal{P}[1..m]\mathcal{T}[1..n]$.

Observe that for any text position t, $1 \leq t \leq n-m+1$, where there is no occurrence of the pattern, there must be at least one text position $\mathcal{W}_t^{\mathcal{T}}$, such that $\mathcal{T}[\mathcal{W}_t^{\mathcal{T}}] \neq \mathcal{P}[\mathcal{W}_t^{\mathcal{T}} - t + 1]$. The position $\mathcal{W}_t^{\mathcal{T}}$ is called a *witness* for the non-occurrence of the pattern at text position t.

The output of the string-matching problem consists of a length n boolean vector whose entries indicate if there are any occurrences of the pattern starting at each of the corresponding text positions. This boolean vector will be encoded the same way as the input strings, with the same parameter λ, and the alphabet symbols 1 and 0. In addition to the boolean vector the algorithm provides witnesses for the non-occurrences of the pattern. Notice that since our algorithm makes only $O(\lceil n/\lambda \rceil)$ operations it is not possible to list all witnesses as in other string-matching algorithms.

The main idea in the new string-matching algorithm is that the witnesses are given implicitly where any specific witnesses can be computed from the output of the algorithm by a single processor in constant time whenever needed. The algorithm is otherwise similar to the parallel string-matching algorithm of Breslauer and Galil [10] with certain modifications that allow it to take advantage of coded strings in order to match short patterns by table lookup.

Theorem 4.1 *The string-matching problem on coded pattern and text strings can be solved in $O(\log\log(m/\lambda))$ time making $O(\lceil n/\lambda \rceil)$ operations and using*

$O(\lceil n/\lambda \rceil)$ space.

We outline the structure of the algorithm next. Initially, there are $n-m+1$ text positions at which an occurrence of the pattern might start. These positions are called *potential occurrences.* Using Lemma 3.2, one can verify in constant time making $O(\lceil m/\lambda \rceil)$ operations if any given potential occurrence is a real occurrence. However, verifying all $O(n)$ potential occurrences this way is too costly if the pattern is long. The strategy followed by most efficient parallel string-matching algorithms first eliminates many potential occurrences and then verifies which of the remaining potential occurrences are real occurrences.

Definition 4.2 *A string $\mathcal{S}[1..k]$ has a period of length p if $\mathcal{S}[i] = \mathcal{S}[i+p]$, for $i = 1, \cdots, k-p$.*

The shortest non-zero period length of a string $\mathcal{S}[1..k]$ is called *the period length* of $\mathcal{S}[1..k]$. Denote by π the period length of the pattern $\mathcal{P}[1..m]$. If p is not a period length of the pattern $\mathcal{P}[1..m]$, then there must exist some pattern position $\mathcal{W}_p^{\mathcal{P}}$, such that $\mathcal{P}[\mathcal{W}_p^{\mathcal{P}}] \neq \mathcal{P}[\mathcal{W}_p^{\mathcal{P}} - p]$. The positions $\mathcal{W}_p^{\mathcal{P}}$ are called *witnesses* for non-periods of the pattern. Notice that the witnesses $\mathcal{W}_p^{\mathcal{P}}$ are defined for all $p = 1, \ldots, \pi - 1$.

Vishkin [38] suggested the *duel* method to eliminate potential occurrences efficiently. His method, which is described next, has been used in all efficient parallel string-matching algorithms afterward as well as in sequential and parallel two-dimensional matching algorithms [1, 13, 17, 25]. The idea in duels is that if there are two potential occurrence of the pattern at positions p and q of the text, such that $0 < q - p < \pi$, then since $\mathcal{P}[\mathcal{W}_{q-p}^{\mathcal{P}}] \neq \mathcal{P}[\mathcal{W}_{q-p}^{\mathcal{P}} - (q-p)]$, the text symbol $\mathcal{T}[p + \mathcal{W}_{q-p}^{\mathcal{P}} - 1]$ can not be equal both to $\mathcal{P}[\mathcal{W}_{q-p}^{\mathcal{P}}]$ and to $\mathcal{P}[\mathcal{W}_{q-p}^{\mathcal{P}} - (q-p)]$. Therefore, text position $p + \mathcal{W}_{q-p}^{\mathcal{P}} - 1$ must be a witness for the non-occurrence of the pattern at text position p or at text position q (possibly at both positions) and the algorithm can eliminate one of the potential occurrences at p or at q by making a single pairwise symbol comparison.

Observe that if the pattern occurs at positions p and q of the text, such that $0 < q - p < m$, then it has a period of length $q - p$ and therefore $\pi \leq q - p$. Thus, there can be no more than n/π occurrences of the pattern in the text. Using duels, it is possible to eliminate efficiently potential occurrences that are close to each other, leaving at most n/π potential occurrences. Still, there might be too many occurrences to verify separately if the period length π is much smaller than the pattern length. In this case the algorithm must follow a different strategy. The algorithm proceeds in few steps:

1. If the pattern length $m \leq 2\lambda$, then the string-matching problem is solved by table lookup as described in Lemma 4.3.

2. If the pattern length $m > 2\lambda$, then the pattern preprocessing step described in Section 4.2 is invoked. It finds the period length of the pattern, π, and the witnesses $\mathcal{W}_p^{\mathcal{P}}$.

(a) If the pattern is found to be non-periodic, namely, if $m \leq 2\pi$, then the algorithms finds the occurrences of the pattern directly, as described in Lemma 4.7.

(b) If the pattern is periodic, namely, if $m > 2\pi$, then the algorithm only searches for occurrences of the non-periodic pattern prefix $\mathcal{P}[1..2\pi]$. This is done as described in Lemma 4.3 if this pattern prefix is short or as described in Lemma 4.7 if it is long.

The algorithm then reconstructs from the occurrences of this pattern prefix and by matching some short pattern suffix, the occurrences of the complete pattern as described in Lemma 4.5.

In the description below we show how the algorithm computes the witnesses $\mathcal{W}_p^{\mathcal{P}}$ for non-periods of the pattern. We do not specify exactly how the witnesses $\mathcal{W}_t^{\mathcal{T}}$ for non-occurrences of the pattern can be computed since their computation is similar to the pattern witnesses and they can be easily reconstructed by tracing the steps of the algorithm.

4.1 Text processing

The saving in the number of processors used by the algorithm is achieved mainly by matching short patterns by table lookup.

Lemma 4.3 *One can find all occurrences of the pattern $\mathcal{P}[1..m]$, such that $m \leq d\lambda$, for some fixed constant $d \geq 1$, in the text $\mathcal{T}[1..n]$, in constant time making $O(\lceil n/\lambda \rceil)$ operations and using $O(\lceil n/\lambda \rceil)$ space.*

Proof: We show how the pattern occurrences can be found making a constant number of operations when the text length $n = m + \lambda - 1$. If the text is longer, then the same procedure is applied simultaneously in overlapping text blocks of length $m + \lambda - 1$, which start λ positions apart, making $O(\lceil n/\lambda \rceil)$ operations.

The algorithm precomputes the lookup table $\mathcal{SM}[\hat{t_1}, \hat{t_2}, \hat{p}, l]$ that gives the answer to the string matching problem with the pattern $\hat{p}$ of length l, $1 \leq l \leq \lambda$, in the text of length $l + \lambda - 1$ that is coded in $\hat{t_1}$ and $\hat{t_2}$. The $\mathcal{SM}$ table provides the coded boolean vector representing all occurrences together with witnesses for all non-occurrences that are represented in an array of size λ. This table requires $O(c^{3\lambda}\lambda^2)$ space.

If the pattern is a longer string that is coded as $\hat{\mathcal{P}}[1..m] = \hat{\mathcal{P}}_1\hat{\mathcal{P}}_2 \cdots \hat{\mathcal{P}}_d$, $(d-1)\lambda < m \leq d\lambda$, and the text is coded as $\hat{\mathcal{T}} = \hat{\mathcal{T}}_1\hat{\mathcal{T}}_2 \cdots \hat{\mathcal{T}}_{d+1}$, then the algorithm solves the string-matching problem by d table lookups. This is done by observing that there is an occurrence of the pattern at position q of the text $\hat{\mathcal{T}}$, $1 \leq q \leq \lambda$, if and only if there are occurrences of $\hat{\mathcal{P}}_i$ at position q of $\hat{\mathcal{T}}_i\hat{\mathcal{T}}_{i+1}$, for all $i = 1, \ldots, d$ ($\hat{\mathcal{P}}_i$'s have length λ except for $\hat{\mathcal{P}}_d$ that might be shorter).

The coded boolean vector representing all occurrences is computed by *masking* the coded representation of the solutions to the d smaller string-matching problems. This can be done efficiently by precomputing the lookup table $\mathcal{MASK}[\hat{a}, \hat{b}]$ that gives the coded boolean vector that represents the occurrences that are represented in both boolean vectors $\hat{a}$ and $\hat{b}$. The witnesses for the non-occurrences will not be combined and when there is a need for a specific witness it can be found in constant time by looking it up in the output of the d smaller string-matching problems sequentially. $\square$

4.1.1 Periodic patterns

In this section we describe how the string-matching algorithm deals with long periodic patterns. Namely $m > \max(2\lambda, 2\pi)$. As mentioned above, in this case the general strategy of eliminating potential occurrences and verifying the remaining ones is too costly since there might be too many real occurrences. The algorithm searches only for occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$, which is non-periodic by the following lemma, and then finds the occurrences of the whole pattern by "counting" consecutive occurrences of this prefix. Recall that the occurrences of $\mathcal{P}[1..2\pi]$ are found by Lemma 4.3, if $\pi \leq \lambda$, and by Lemma 4.7 otherwise.

Lemma 4.4 *(Lyndon and Schutzenberger [30]) If a string of length k has two periods of lengths p and q and $p + q \leq k$, then it also has a period of length* $\gcd(p, q)$*.*

Breslauer and Galil [10] suggested the following method to find occurrences of the full pattern given the occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$. Assume without loss of generality that the text length $n \leq 3m/2$. Call an occurrence of the pattern prefix $\mathcal{P}[1..2\pi]$ at text position i an *initial occurrence* if there is no occurrence of this prefix at position $i - \pi$ and a *final occurrence* if there is no occurrence of this prefix at position $i + \pi$. Let $\mathcal{I}$ be the largest initial occurrence in the first $m/2$ positions of the pattern and let $\mathcal{F}$ be the smallest final occurrence that is larger than $\mathcal{I}$. It is not difficult to verify that the only occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$ that are occurrences also of the entire pattern are those between positions $\mathcal{I}$ and $\mathcal{F} - \pi * (\lfloor m/\pi \rfloor - 3)$ and possibly also the occurrence at position $\mathcal{F} - \pi * (\lfloor m/\pi \rfloor - 2)$ if there is an occurrence of the pattern prefix $\mathcal{P}[1..l]$, $l = m - \pi * \lfloor m/\pi \rfloor$, at position $\mathcal{F} + 2\pi$.

Lemma 4.5 *Given the occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$ in the text $\mathcal{T}[1..n]$, it is possible to find the occurrences of the entire pattern in constant time making $O(n/\lambda)$ operations and using $O(n/\lambda)$ space.*

Proof: Recall that $n \leq 3m/2$. If the pattern period $\pi < \lambda$, then the initial and final occurrences are found by the lookup tables $\mathcal{INIT}[\hat{t}_1, \hat{t}_2, \pi]$ and $\mathcal{FINAL}[\hat{t}_1, \hat{t}_2, \pi]$ that give for the boolean vectors $\hat{t}_1$ and $\hat{t}_2$ that represent the occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$, the boolean vectors representing only the initial or final occurrences, respectively. If the pattern period $\pi \geq \lambda$, then the occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$ must be spread at least π positions apart from each other and the initial and final occurrences are found by examining for each occurrence of the pattern prefix $\mathcal{P}[1..2\pi]$ if there is an occurrence π position before and after it. In both cases the initial and final occurrences can be clearly found in constant time and $O(n/\lambda)$ operations.

The important initial and final occurrences $\mathcal{I}$ and $\mathcal{F}$ are then found similarly to Lemma 3.2. Using $\mathcal{I}$ and $\mathcal{F}$ and after verifying if there is an occurrence of the pattern prefix $\mathcal{P}[1..l]$, $l = m - \pi * \lfloor m/\pi \rfloor$, at position $\mathcal{F} + 2\pi$, by Lemma 3.2, the algorithm knows which occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$ are actually occurrences of the whole pattern. Notice that the output boolean vector representing the occurrences of the pattern can be created efficiently since these occurrences are a contiguous subset of the occurrences of the pattern prefix $\mathcal{P}[1..2\pi]$. Thus, the whole computation takes constant time, makes $O(n/\lambda)$ operations and uses $O(n/\lambda)$ space. □

4.1.2 Non-periodic patterns

In this section we describe how the string-matching algorithm deals with long non-periodic patterns. Namely $2\lambda < m \leq 2\pi$ and therefore $\pi > \lambda$.

Lemma 4.6 *If the pattern $\mathcal{P}[1..m]$ has period length $\pi \geq \lambda$, then it contains a substring $\mathcal{P}[z..z + 2\lambda - 1]$, called a* synchronizing block*, with period length that is at least λ.*

Proof: Recall that $m > 2\lambda$. Let $\hat{\pi}$ be the period length of the pattern prefix $\mathcal{P}[1..2\lambda]$. If $\hat{\pi} \geq \lambda$, then this prefix is the required substring. Otherwise, let $\mathcal{P}[1..l]$ be the longest prefix of the pattern whose period length is $\hat{\pi}$. By Lemma 4.4, the period length of $\mathcal{P}[l - 2\lambda + 2..l]$ is also $\hat{\pi}$ and the period length of $\mathcal{P}[l - 2\lambda + 2..l + 1]$ is at least λ. □

The pattern preprocessing described in the next section computes the period length of the pattern, the witnesses $\mathcal{W}_p^{\mathcal{P}}$ and a synchronizing block which are used in the next lemma.

Lemma 4.7 *The string matching problem with the coded pattern $\mathcal{P}[1..m]$ and text $\mathcal{T}[1..n]$, such that $2\lambda < m \leq 2\pi$, is solved in $O(\log\log(m/\lambda))$ time making $O(n/\lambda)$ operations and using $O(n/\lambda)$ space.*

Proof: The algorithm starts eliminating potential occurrences by finding all occurrences of the synchronizing block $\mathcal{P}[z..z + 2\lambda - 1]$ in the text using the table lookup in Lemma 4.3. Observe that there might be an occurrence of the pattern at text position q only if there is an occurrence of the synchronizing block $\mathcal{P}[z..z + 2\lambda - 1]$ at text position $q + z - 1$. Since the period length of the synchronizing block is at least λ, the remaining potential occurrences must be spaced at least λ positions apart and there can be at most $\lceil n/\lambda \rceil$ potential occurrences left. Namely, at most one potential occurrence left within each coded word representing the text. The positions of the remaining potential occurrences are written into an array of size $O(n/\lambda)$. Notice that the witnesses for the non-occurrences of the potential occurrences eliminated in this step are given implicitly by matching the synchronizing block. The other witnesses that are computed later will be stored explicitly in an array.

The elimination of the remaining potential occurrences continues as in the algorithm of Breslauer and Galil [10]. Notice that, for technical reasons, the pattern preprocessing step computes the witnesses $\mathcal{W}_p^{\mathcal{P}}$, only for $p = 1, \ldots, \lceil m/2 \rceil$.

The algorithm first partitions the text into consecutive blocks of length $\lambda \log\log(m/\lambda)$. There are at most $\log\log(m/\lambda)$ potential occurrences left in each such block. By performing duels, the algorithm eliminate all but at most one potential occurrence in each block. This takes $O(\log\log(m/\lambda))$ time using a single processor per block. The entire computation makes $O(n/\lambda)$ operations.

The algorithm then partitions the text into blocks of length $\lceil m/2 \rceil$ and proceed in each block simultaneously using $m/\lambda \log\log(m/\lambda)$ processors per block. In each block there are at most $m/\lambda \log\log(m/\lambda)$ potential occurrences left. The algorithm recursively partitions blocks with h potential occurrences into $\sqrt{h}$ blocks with $\sqrt{h}$ potential occurrences, giving $\sqrt{h}$ processors to handle each block. The recursive step leaves at most one potential occurrence in each of the $\sqrt{h}$ blocks. Then, using h processors for performing duels between all pairs of the remaining $\sqrt{h}$ potential occurrences in the block, the algorithm eliminates all but one potential occurrence in the block. The depth of the recursion, which is the time spent, is $O(\log\log(m/\lambda))$.

After the elimination of potential occurrences described above there are at most $O(n/m)$ potential occurrences left. The algorithm verifies these potential occurrences to be real occurrences using Lemma 3.2. The entire computation takes $O(\log\log(m/\lambda))$ time making $O(n/\lambda)$ operations and using $O(n/\lambda)$ space. $\square$

4.2 Pattern preprocessing

The pattern preprocessing is invoked only if $m > 2\lambda$. It has to find the period length π of the pattern and the witnesses $\mathcal{W}_p^{\mathcal{P}}$. For technical reasons, the pattern preprocessing step computes only the witnesses $\mathcal{W}_p^{\mathcal{P}}$, for $p = 1, \ldots, \min(\lceil m/2 \rceil, \pi - 1)$. In addition, if $\pi \geq \lambda$, then the pattern preprocessing step finds also a synchronizing block.

Notice, that if the period length of the pattern $\pi > \lceil m/2 \rceil$, then it is not computed precisely. In this case the pattern is non-periodic and the period length π is not used by the algorithm.

Lemma 4.8 *The pattern preprocessing step with the coded pattern $\mathcal{P}[1..m]$, such that $m > 2\lambda$, takes $O(\log\log(m/\lambda))$ time making $O(m/\lambda)$ operations and using $O(m/\lambda)$ space.*

Proof: The pattern preprocessing step first finds a synchronizing block and then uses this block and witnesses that it has already computed to compute more witnesses in iterations that resemble the text processing step. The indices p for which the witnesses $\mathcal{W}_p^{\mathcal{P}}$ are not yet computed are called *potential period lengths.* The witnesses $\mathcal{W}_p^{\mathcal{P}}$, $p = 1, \ldots, \min(\lceil m/2 \rceil, \pi - 1)$, will be given implicitly, where any specific witness can be produced from the information computed in constant time by a single processor.

The pattern preprocessing uses a precomputed lookup table, similarly to the $\mathcal{SM}$ table from Lemma 4.3, that gives the boolean vector representing the *period lengths* and the witnesses for the non-periods of a short string. If the pattern length $m \leq 4\lambda$, then the pattern preprocessing step will be solved directly by this table lookup[2]. Thus, from here on we assume that the pattern length $m > 4\lambda$.

Our first goal is to find a synchronizing block and to reduce the number of potential period lengths to $O(m/\lambda)$. Recall the constructive nature of the proof of Lemma 4.6. Using the precomputed table of period lengths of short strings, the algorithm finds the period length $\hat{\pi}$ of the pattern prefix $\mathcal{P}[1..2\lambda]$. If $\hat{\pi} \geq \lambda$, then the algorithm has found the synchronizing block $\mathcal{P}[1..2\lambda]$. Otherwise, if $\hat{\pi} < \lambda$, the algorithm checks if the whole pattern has period length $\hat{\pi}$, by Lemma 3.2. If $\hat{\pi}$ turns out to be the period length of the whole pattern, then the only information required from the pattern preprocessing step is this period length $\pi = \hat{\pi}$, and the pattern preprocessing is completed. Otherwise, the synchronizing block $\mathcal{P}[z..z + 2\lambda - 1]$ has been found.

If $z + \lambda - 1 > \lceil m/2 \rceil$, then by the construction of the synchronizing block in Lemma 4.6, the pattern prefix $\mathcal{P}[1..z + 2\lambda - 2]$ has period length $\hat{\pi}$ and $\mathcal{P}[1..z + 2\lambda - 1]$ does not have this period length. Thus, by Lemma 4.4, full occurrences of the pattern prefix $\mathcal{P}[1..2\lambda]$ that start in the first $\lceil m/2 \rceil$ positions of the pattern, start at positions $k\hat{\pi} + 1$. Matching the pattern prefix $\mathcal{P}[1..2\lambda]$ by Lemma 4.3, one obtains the witnesses $\mathcal{W}_p^{\mathcal{P}}$, except for the multiples $p = k\hat{\pi}$. The position $z + 2\lambda - 1$ where the period of length $\hat{\pi}$ terminates provides the witness $\mathcal{W}_{\hat{\pi}}^{\mathcal{P}}$, and since the pattern prefix $\mathcal{P}[1..z + 2\lambda - 2]$ has period length $\hat{\pi}$, $\mathcal{W}_p^{\mathcal{P}} = z + 2\lambda - 1$, for all the multiples $p = k\hat{\pi}$, such that $0 < p \leq \lceil m/2 \rceil$. Thus, the witnesses $\mathcal{W}_p^{\mathcal{P}}$ can be reconstructed either by matching the pattern prefix $\mathcal{P}[1..2\lambda]$, by Lemma 4.3, if p is not a multiple of $\hat{\pi}$, or $\mathcal{W}_p^{\mathcal{P}} = z + 2\lambda - 1$ otherwise.

If $z + \lambda - 1 \leq \lceil m/2 \rceil$, then the algorithm finds all occurrences of the synchronizing block $\mathcal{P}[z..z+2\lambda-1]$ in the pattern, by Lemma 4.3. Observe that the witness to the non-occurrence of the synchronizing block at pattern position $p + z$ correspond to the witness $\mathcal{W}_p^{\mathcal{P}}$. The occurrences, which must be spaced at least λ positions apart, leave at most $O(m/\lambda)$ potential period lengths in the first half of the pattern. (This is not completely true. If $\lceil m/2 \rceil \leq z + 2\lambda - 1 \leq \lceil m/2 \rceil + \lambda$, then there can be no occurrences of the synchronizing block at positions that are larger than or equal to $m - z - 2\lambda$. However, it is possible to achieve the goal by searching for occurrences of the pattern prefix $\mathcal{P}[1..2\lambda]$ at position $m - z - 2\lambda, \ldots, \lceil m/2 \rceil$.) The positions of the remaining potential period lengths are written into an array and their witnesses will be computed and stored explicitly as we show next. Observe that when a specific witness is called for, it can be either reconstructed by matching the synchronizing block again or it will be stored explicitly in a table.

The computation of the remaining witnesses proceedings in the same fashion as the string-matching algorithm of Breslauer and Galil [10]. We sketch here only a non-optimal version of the algorithm making $O(m \log\log(m/\lambda)/\lambda)$ operations. The algorithm can be made optimal similarly to the algorithm of Breslauer and Galil.

[2]An alternative implementation would match these short patterns by the table lookup in Lemma 4.3. This would reduce the size of the lookup table we use here to find the period lengths of short strings, but would not eliminate completely the need for this lookup table, since this table is still used later to find the period length of the pattern prefix $\mathcal{P}[1..2\lambda]$.

The algorithm proceeds in iterations and maintains the invariant that at the beginning of iteration number i, there is at most one potential period length (yet-to-be-computed witness) in each block of length k_i, where,

$$k_i = m^{1-\frac{1}{2^i}} \cdot \lambda^{\frac{1}{2^i}} \quad \text{for} \quad i = 0, \ldots, \log\log(m/\lambda).$$

Clearly, the invariant holds at the beginning of iteration number 0, since the potential period lengths remaining after the first part of the computation are spaced at least $k_0 = \lambda$ positions apart.

At the beginning of iteration number i, there are at most k_{i+1}/k_i potential period length in each block of length k_{i+1}. The algorithm checks using Lemma 3.2, which of the potential period lengths in the first k_{i+1} block is a period length of the pattern prefix $\mathcal{P}[1..2k_{i+1}]$. Those potential period length which are eliminated have their witness determined, while the remaining potential period lengths, if any, are multiples of the shortest remaining period length, by Lemma 4.4. This computation takes constant time and $O(k_{i+1}/\lambda)$ operations for each potential period, or $O(k_{i+1}^2/k_i\lambda) = O(m/\lambda)$ operations in total.

If there are any potential period lengths remaining in the first k_{i+1} block, then the algorithm verifies whether the shortest one is the period length of the whole pattern by Lemma 3.2. If it is found to be the period length then the computation is complete.

Otherwise, the smallest position at which this periodicity is terminated is a witness for all multiples of the shortest period in the first k_{i+1} block. Now, it remains only to eliminate all but at most one potential period length in each k_{i+1} block, before proceeding to the next iteration.

It is possible to eliminate all but at most one potential period length in each k_{i+1} block using duels, since at this point we have the witnesses $\mathcal{W}_p^{\mathcal{P}}$, for all $p = 1, \ldots, k_{i+1}$. The duels, however, are slightly different from those used in the text processing step, since occurrences might be overhanging: a duel that has to produce one of the witnesses $\mathcal{W}_i^{\mathcal{P}}$ or $\mathcal{W}_j^{\mathcal{P}}$, for $i < j < \lceil m/2 \rceil$, will normally produce the witness $i + \mathcal{W}_{j-i}^{\mathcal{P}} + 1$, if it is within the pattern; otherwise the duel produces the witnesses $\mathcal{W}_i^{\mathcal{P}} = \mathcal{W}_{j-i}^{\mathcal{P}} - j + i$ or $\mathcal{W}_j^{\mathcal{P}} = \mathcal{W}_{j-i}^{\mathcal{P}}$.

The duels are carried out in the same fashion as in the text processing step. However, we allow the algorithm to use $m/\lambda \log\log(m/\lambda)$ processors. The duels will take at most $O(\log\log(m/\lambda))$ time in the first two iterations of the pattern preprocessing, after which they take constant time since the number of remaining potential period lengths will be small enough relatively to the number of available processors.

The whole pattern preprocessing step described above takes $O(\log\log(m/\lambda))$ time. The overall number of operations used is $O(m/\lambda)$ except at the step that verifies if the shortest remaining potential period length in each iteration is the period length of the whole pattern. This step uses $O(m/\lambda)$ operation in each iteration and thus $O(m \log\log(m/\lambda)/\lambda)$ operations over all iteration. However, this step can be implemented more economically, making only $O(m/\lambda)$ operations [10]. $\Box$

5 Applications

In this section we present two application of the string-matching algorithm described above in reducing the number of processors used in known parallel algorithms for testing if a string is square-free and for finding all palindromes in a string. The reduction in the number of processors is achieved since the input string $\mathcal{S}[1..n]$ has to be encoded only once while its encoded substrings are presented several times as input to the string-matching algorithm. Recall that the input string $\mathcal{S}[1..n]$ is encoded with the parameter $\lambda = O(\log n)$.

5.1 Testing if a string is square-free

A non-empty string of the form xx is called a repetition. A *square* is defined as a repetition xx, where x is primitive, or in other words $x \neq v^h$ for all strings v and integers $h > 1$. Strings that do not contain any substring that is a repetition are called *repetition-free* or *square-free*. For example '*aa*', '*abab*' and '*baba*' are the repetitions which are contained in the string '*baababa*'. It is not difficult to verify that any string with at least four symbols over alphabets with two symbols contains a square. However, there exist infinite length strings on three letter alphabets that are square-free as shown by Thue [36, 37].

In the sequential setting, algorithms for testing if a string is square-free and for finding all repetitions in a string were designed by Apostolico and Preparata [7], Crochemore [14, 15], Kosaraju [28], Main and Lorentz [31, 32] and Rabin [34]. Main and Lorentz [31] proved that it is possible to find all repetition in a string in $O(n \log n)$ time using pairwise comparison of input symbols that test for equality. They have also shown that $\Omega(n \log n)$ equality tests are necessary even to decide if a string is square-free. Main and Lorentz [32] have shown using the "four Russians technique" that if the input alphabet has constant size, then it is possible to test if a string is square-free in $O(n)$ time. The same bound was obtained by Crochemore [15] using a different method. Notice that it is not possible to list all squares in $O(n)$ time since there might be too many squares [2, 14].

In the parallel setting, Crochemore and Rytter [18, 19] test if a string is square-free in $O(\log n)$ time using n processors and $O(n^{1+\epsilon})$ space. Apostolico [3] designed an algorithm that tests if a string is square-free and also detects all squares within the same time and processor bounds using only linear auxiliary space. If the input alphabet has constant size, then Apostolico's algorithm can use the "four Russians technique" to tests if a string is square-free in $O(\log n)$ time utilizing only $n/\log n$ processors.

Apostolico and Breslauer [4] gave a parallel implementation of the sequential algorithm of Main and Lorentz [32] to test if a string is square-free and find all square in a string using equality tests in $O(\log\log n)$ time using $n \log n/\log\log n$ processors. If the input alphabet has constant size, then the number of processors used by their algorithm to test if a string is square-free can be reduced to $n/\log\log n$ by using the new string-matching algorithm. These bounds compare favorably also with the $O(\log n)$ time algorithm given by Apostolico [3] for testing if a string over a constant size alphabet is square-free. Notice that all the parallel algorithms mentioned above achieve an optimal speedup since their time-processor product is the same as the time complexity of the fastest known sequential algorithm under the same assumptions on the input alphabet.

Theorem 5.1 *There exists an algorithm to test if a string $\mathcal{S}[1..n]$ over a constant size alphabet is square-free in $O(\log\log n)$ time using $n/\log\log n$ processors and $O(n)$ space.*

The details of the algorithm can be found in Apostolico and Breslauer's paper [4]. The necessary modifications to take advantage of the coded strings are similar to- and simpler than those of the palindrome detection algorithm that is discusses in more details next.

5.2 Finding all palindromes in a string

Palindromes are symmetric strings that read the same forward and backward. Formally, a non-empty string w is a palindrome if $w = w^R$, where w^R denotes the string w reversed. It is convenient to distinguish between even length palindromes that are strings of the form $w = vv^R$ and odd length palindromes that are strings of the form $w = vav^R$, where v is an arbitrary string and 'a' is a single alphabet symbol.

Given a string $\mathcal{S}[1..n]$, we say that there is an even palindrome of radius $\mathcal{R}$ *centered at* position k of $\mathcal{S}[1..n]$, if $\mathcal{S}[k-i] = \mathcal{S}[k+i-1]$, for $i = 1, \ldots, \mathcal{R}$. We say that there is an odd palindrome of radius $\hat{\mathcal{R}}$ *centered on* position k of $\mathcal{S}[1..n]$, if $\mathcal{S}[k-i] = \mathcal{S}[k+i]$, for $i = 1, \ldots, \hat{\mathcal{R}}$. The radius $\mathcal{R}$ (or $\hat{\mathcal{R}}$) is maximal if there is no palindrome of radius $\mathcal{R}+1$ centered at (on) the same position. In this section we will be interested in computing the maximal radii $\mathcal{R}[k]$ and $\hat{\mathcal{R}}[k]$ of the even and the odd palindromes which are centered at (on) all positions k of $\mathcal{S}[1..n]$. Notice that if we double each input symbol, then odd palindromes become even and thus, without loss of generality, we can concentrate on finding only the maximal radii of the even palindromes [6].

In the sequential setting, Manacher [33], and Knuth, Morris and Pratt [27] presented linear-time algorithms that find the initial palindromes (palindrome prefixes) of a string. Galil [22] and Slisenko [35] presented real-time algorithms on multi-tape Turing machines to find all initial palindromes. A closer look at Manacher's algorithm reveals that it not only finds the initial palindromes, but it also computes the maximal radii of palindromes centered at all positions of the input string using pairwise symbol comparisons that test for equality. Thus it solves the problem we consider in this section in $O(n)$ time. Notice that although the similarity between the definitions of squares and palindromes is obvious, the computational complexities of detecting squares and palindromes using equality tests are inherently different. The parallel algorithms discussed in this paper, however, are quite similar.

In the parallel setting, Crochemore and Rytter [19] presented an algorithm that finds all palindromes in a string in $O(\log n)$ time using n processors and $O(n^{1+\epsilon})$ space. Their algorithm assumes that the alphabet symbols are small integers. Breslauer and Galil [12], using an observation of Fischer and Paterson

[21], described an algorithm that finds all initial palindromes in a string in $O(\log\log n)$ time and $n/\log\log n$ processors using equality tests.

Apostolico, Breslauer and Galil [6] gave an algorithm that can find all palindromes in a string using equality tests in $O(\log\log n)$ time and $n\log n/\log\log n$ processors. They also gave an optimal-speedup algorithm that finds all palindromes in a string over constant size alphabets in $O(\log n)$ time and $n/\log n$ processors, using the "four Russians technique". We show next that if the input alphabet has constant size then the number of processors used in their $O(\log\log n)$ time algorithm can be reduces to $n/\log\log n$, achieving an optimal speedup.

Theorem 5.2 *There exists an algorithm that finds all even palindromes in a string $\mathcal{S}[1..n]$ over a constant size alphabet in $O(\log\log n)$ time using $n/\log\log n$ processors and $O(n)$ space.*

We outline the main parts of the algorithm of Apostolico, Breslauer and Galil [6] and point out where we take advantage of coded strings. The missing proofs and a more complete description of the algorithm can be found in Apostolico, Breslauer and Galil's paper. Notice that the algorithm sometimes refers to reversed substrings, and thus we have to encode both the original input string and its reverse. Alternatively, we can precompute a table that will provide for each coded block of symbols, the coded representation of the reversed block. To simplify the presentation, assume without loss of generality that the algorithms can access symbols whose indices are out of the boundaries of the input string. These symbols are considered to be different from each other and from the symbols of $\mathcal{S}[1..n]$.

The main observation that allows to find the radii of many palindromes together is given in the following lemma.

Lemma 5.3 *Assume that the string $\mathcal{S}[1..n]$ contains an even palindrome whose radius is at least r centered at position p. Furthermore, let $\mathcal{S}[\epsilon_L..\epsilon_R]$ be the maximal substring that contains $\mathcal{S}[p-r..p+r-1]$ and is periodic with period length $2r$. Namely, $\mathcal{S}[i]=\mathcal{S}[i+2r]$, for $i=\epsilon_L,\ldots,\epsilon_R-2r$, and $\mathcal{S}[\epsilon_L-1]\neq\mathcal{S}[\epsilon_L+2r-1]$ and $\mathcal{S}[\epsilon_R+1]\neq\mathcal{S}[\epsilon_R-2r+1]$.*

Then the maximal radii of the palindromes centered at positions $q=p+lr$, for integral positive or negative values of l, such that $\epsilon_L\leq q\leq\epsilon_R$, are given as follows:

- *If $q-\epsilon_L\neq\epsilon_R-q+1$, then the radius is exactly $\min(q-\epsilon_L,\epsilon_R-q+1)$.*
- *If $q-\epsilon_L=\epsilon_R-q+1$, then the radius is larger than or equal to $q-\epsilon_L$. The radius is exactly $q-\epsilon_L$ if and only if $\mathcal{S}[\epsilon_L-1]\neq\mathcal{S}[\epsilon_R+1]$.*

The algorithm proceeds in independent stages which are computed simultaneously. In stage number η, $0\leq\eta\leq\lfloor\log_2 n\rfloor-3$, the algorithm computes all entries $\mathcal{R}[i]$ of the radii array such that $4l_\eta\leq\mathcal{R}[i]<8l_\eta$, for $l_\eta=2^\eta$. Notice that each stage computes disjoint ranges of the radii values and that all possible radii values are computed by some stage.

The remainder of this section describes a generic stage number η. Partition the input string $\mathcal{S}[1..n]$ into consecutive blocks of length l_η. Stage number η consists of independent sub-stages that are assigned to each such block and computed simultaneously. Each sub-stage finds the radii of all palindromes which are centered in the block that it is assigned to and whose radii are in the range computed by stage η. Sometimes palindromes whose radii are out of this range can be detected, but these radii do not have to be written into the output array since they are guaranteed to be found in an other stage.

The sub-stage that is assigned to block number h starts with a call to the string-matching algorithm to find all occurrences of the four consecutive blocks $\mathcal{S}[(h-4)l_\eta + 1..hl_\eta]$, reversed, in $\mathcal{S}[(h-2)l_\eta + 1..(h+4)l_\eta - 1]$. Let $p_1 < p_2 < \cdots < p_r$ denote the indices of all these occurrences. The next lemma states that we essentially found all "interesting" palindromes.

Lemma 5.4 *There exists a correspondence between the elements of the $\{p_i\}$ sequence and all palindromes that are centered in block number h and whose radii are large enough.*

- *If $p_i + hl_\eta$ is odd, then p_i corresponds to an even palindrome which is centered at position $(p_i + hl_\eta + 1)/2$.*
- *If $p_i + hl_\eta$ is even, then p_i corresponds to an odd palindrome which is centered on position $(p_i + hl_\eta)/2$.*

Each palindrome whose radius is at least $4l_\eta - 1$ has some corresponding p_i, while palindromes that correspond to some p_i are guaranteed to have radii that are at least $3l_\eta$.

Lemma 5.5 *The sequence $\{p_i\}$, which is defined above, forms an arithmetic progression.*

By the last lemma the sequence $\{p_i\}$ can be represented by three integers: the start, the difference and the sequence length. This representation can be computed from the output of the string matching algorithm in constant time and $O(\lceil l_\eta/\lambda \rceil)$ operations since it suffices to find the positions of the first, second and last occurrences. Define the sequence $\{q_i\}$, for $i = 1, \ldots, l$, to list all centers of the even palindromes that correspond to elements in $\{p_i\}$. By Lemma 5.4, the sequence $\{q_i\}$ also forms an arithmetic progression and therefore it can also be computed and manipulated efficiently.

If the $\{q_i\}$ sequence does not contain any element, then there are no even palindromes whose radius is at least $4l_\eta$ that are centered in the current block. If there is only one element q_1, then by Lemma 3.2, we can find in constant time and $O(\lceil l_\eta/\lambda \rceil)$ operations what is the radius of the palindrome that is centered at q_1 or we can conclude that it is too large to be computed in this stage. If there are more elements, let q denote the difference of the arithmetic progression $\{q_i\}$. The next lemma shows how to find the radii of the palindromes centered at $\{q_i\}$ efficiently.

Lemma 5.6 *It is possible to find the radii of all even palindromes centered at positions in $\{q_i\}$, which are in the range that is computed in this stage, in constant time and $O(\lceil l_\eta/\lambda \rceil)$ operations.*

Proof: Let ζ_L be the smallest index such that $q_1 - 8l_\eta \le \zeta_L < q_1$ and $\mathcal{S}[\zeta_L..q_1 - 1] = \mathcal{S}[\zeta_L + 2q..q_1 + 2q - 1]$, and ζ_R be the largest index such that $q_l \le \zeta_R < q_l + 8l_\eta$ and $\mathcal{S}[q_l - 2q..\zeta_R - 2q] = \mathcal{S}[q_l..\zeta_R]$. The indices ζ_L and ζ_R are computed in constant time and $O(\lceil l_\eta/\lambda \rceil)$ operations by Lemma 3.2. By Lemma 5.3, the radius of the palindrome centered at position q_i is at least $\rho_i = \min(q_i - \zeta_L, \zeta_R - q_i + 1)$. If $\rho_i \ge 8l_\eta$, then the radius of the palindrome centered at q_i is too large to be computed in this stage and it does not have to be determined exactly. Otherwise, the radius is exactly ρ_i except for at most one of the q_i's which satisfies $q_i - \zeta_L = \zeta_R - q_i + 1$. For this particular q_i, by Lemma 3.2, we can find in constant time and $O(\lceil l_\eta/\lambda \rceil)$ operations what is the radius of the palindrome or we can conclude that it is too large to be computed in this stage. $\Box$

The number of radii that are computed in some given sub-stage can be as large as $O(l_\eta)$. This might cause a scheduling problem since even if the overall algorithm can make enough operations to update the whole radii array, it can not make more than $O(\lceil l_\eta/\lambda \rceil)$ operations in the given sub-stage. To overcome this problem we agree that the algorithm will output only few representatives for each group of radii that are found in the same sub-stage. These representative will contain enough information to reconstruct the radii of all palindromes later.

The algorithm partitions the output array $\mathcal{R}[h]$ into contiguous blocks of length λ. When some palindromes are discovered, it writes only one representative for each palindrome group per each block. The representative will contain a description of the part of the $\{q_i\}$ sequence that falls within the block together with ζ_L and ζ_R. Thus, the algorithm does not write more than $O(\lceil l_\eta/\lambda \rceil)$ representatives.

After all stages and sub-stages are completed, in each λ-block of the output array $\mathcal{R}[k]$, the number of palindromes to be reconstructed from the representatives is counted. This can be done in $O(\log \lambda)$ time using $\lambda / \log \lambda$ processors per block by Lemma 2.1. Then, the λ processors that are available in each block of length λ can be properly assigned to create the complete output array with the radii of all palindromes.

Proof of Theorem 5.2: Stage number η has $\lfloor n/l_\eta \rfloor$ sub-stages. Each sub-stage solves a string-matching problem and then by Lemma 5.6, it finds the palindromes that correspond to the occurrences discovered. Thus, each sub-stage takes $O(\log\log(l_\eta/\lambda))$ time and makes $O(\lceil l_\eta/\lambda \rceil)$ operations using $O(\lceil l_\eta/\lambda \rceil)$ space. Therefore, stage number η takes $T_\eta = O(\log\log(l_\eta/\lambda))$ time and makes $O(\lceil l_\eta/\lambda \rceil \times \lfloor n/l_\eta \rfloor)$ operations using $O(\lceil l_\eta/\lambda \rceil \times \lfloor n/l_\eta \rfloor)$ space.

Recall that $\lambda = O(\log n)$. The algorithm takes $\max T_\eta = O(\log\log n)$ time. In all the $\log n$ stages, the algorithm makes $O(n)$ operations and uses $O(n)$ space. The last step that reconstructs all entries of the output radii array from their representatives also takes $O(\log\log n)$ time making $O(n)$ operations and using $O(n)$ space. $\Box$

6 Conclusions

The string-matching algorithm presented in this paper takes advantage of the bounded alphabet size to reduce the number of processor used. Since the lower bound of Breslauer and Galil [11, 12] does not hold if the alphabet has constant size, one can hope to design an optimal-speedup algorithms for several string problems, such as the string-matching, the square-detection and the palindrome-detection problems, that will achieve faster running times over constant size alphabets.

An other interesting open question remaining is whether there exists a fast optimal-speedup palindrome detection algorithm using only pairwise symbol comparisons.

References

[1] A. Amir, G. Benson, and M. Farach. An alphabet-independent approach to two-dimensional pattern-matching. *SIAM J. Comput.*, 23(2):313–323, 1994.

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

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

[4] A. Apostolico and D. Breslauer. An Optimal $O(\log\log n)$ Time Parallel Algorithm for Detecting all Squares in a String. Technical Report CUCS-040-92, Computer Science Dept., Columbia University, 1992.

[5] 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.

[6] A. Apostolico, D. Breslauer, and Z. Galil. Parallel Detection of all Palindromes in a String. *Theoret. Comput. Sci.*, 1994. To appear.

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

[8] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

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

[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*, 1994. To appear.

[13] R. Cole, M. Crochemore, Z. Galil, L. Gąsieniec, R. Hariharan, S. Muthukrishnan, K. Park, and W. Rytter. Optimally fast parallel algorithms for preprocessing and pattern matching in one and two dimensions. In *Proc. 34th IEEE Symp. on Foundations of Computer Science*, pages 248–258, 1993.

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

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

[16] M. Crochemore, Z. Galil, L. Gąsieniec, K. Park, and W. Rytter. Constant-Time Randomized Parallel String Matching. Manuscript, 1994.

[17] M. Crochemore, L. Gąsieniec, R. Hariharan, S. Muthukrishnan, and W. Rytter. A Constant Time Optimal Parallel Algorithm for Two Dimensional Pattern Matching. Manuscript, 1993.

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

[19] 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.

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

[21] M.J. Fischer and M.S. Paterson. String matching and other products. In R.M. Karp, editor, *Complexity of Computation*, pages 113–125. American Mathematical Society, Prividence, RI., 1974.

[22] Z. Galil. Palindrome Recognition in Real Time by a Multitape Turing Machine. *J. Comput. System Sci.*, 16(2):140–157, 1978.

[23] Z. Galil. Optimal parallel algorithms for string matching. *Inform. and Control*, 67:144–157, 1985.

[24] Z. Galil. A Constant-Time Optimal Parallel String-Matching Algorithm. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 69–76, 1992.

[25] Z. Galil and K. Park. Truly Alphabet-Independent Two-Dimensional Pattern Matching. In *Proc. 33th IEEE Symp. on Foundations of Computer Science*, pages 247–256, 1992.

[26] T. Goldberg and U. Zwick. Faster parallel string matching via larger deterministic samples. *J. Algorithms*, 16(2):295–308, 1994.

[27] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322–350, 1977.

[28] S.R. Kosaraju. Computation of Squares in a String. In *Proc. 5rd Symp. on Combinatorial Pattern Matching*, number 807 in Lecture Notes in Computer Science, pages 146–150. Springer-Verlag, Berlin, Germany, 1994.

[29] R.E. Lander and M.J. Fischer. Parallel prefix computation. *J. Assoc. Comput. Mach.*, 27(4):831–838, 1980.

[30] 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.

[31] 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.

[32] 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.

[33] G. Manacher. A new Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *J. Assoc. Comput. Mach.*, 22:346–351, 1975.

[34] 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.

[35] A.O. Slisenko. Recognition of palindromes by multihead Turing machines. In V.P. Orverkov and N.A. Sonin, editors, *Problems in the Constructive Trend in Mathematics VI (Proceedings of the Steklov Institute of Mathematics, No. 129)*, pages 30–202. Academy of Sciences of the USSR, 1973. English Translation by R.H. Silverman, pp. 25–208, Amer. Math. Soc., Providence, RI, 1976.

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

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

[38] U. Vishkin. Optimal parallel pattern matching in strings. *Inform. and Control*, 67:91–113, 1985.

[39] U. Vishkin. Deterministic sampling - a new technique for fast pattern matching. *SIAM J. Comput.*, 20(1):22–40, 1990.