Efficient String Algorithmics

(Ph.D. Thesis)

Dany Breslauer

CUCS-024-92

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

1992

Efficient String Algorithmics

Dany Breslauer

ABSTRACT

Problems involving strings arise in many areas of computer science and have numerous practical applications. We consider several problems from a theoretical perspective and provide efficient algorithms and lower bounds for these problems in sequential and parallel models of computation.

In the sequential setting, we present new algorithms for the string matching problem improving the previous bounds on the number of comparisons performed by such algorithms.

In parallel computation, we present tight algorithms and lower bounds for the string matching problem, for finding the periods of a string, for detecting squares and for finding initial palindromes.

Contents

List of Figures

ACKNOWLEDGMENTS

My advisor, Zvi Galil, deserves special thanks for putting up with me all these years. I have learned a lot from his enthusiastic attitude toward work.

I would like also to thank the other members of my thesis defense committee: Terry Boult, Gadi Landau, Neil Sarnak and Thanasis Tsantilas; Terry Boult and Thanasis Tsantilas for serving on my area paper committee and Thanasis Tsantilas and Henryk Wozniakowski for serving on my thesis proposal committee.

My co-authors on the papers described in this thesis provided me with much help and advise; especially, Alberto Apostolico, Omer Berkmen, Baruch Scheiber and Uzi Vishkin. Much of the work in this thesis was done while I was visiting Alberto at Università de L'Aquila, L'Aquila, Italy.

I have spent very special three and a half years at Columbia studying for this degree. There are many friends whom I would like to thank for making my stay in New York a very pleasant one. I try to mention some of them below.

Pino Italiano taught me many pasta recipes and facts of life. Matt Franklin tried to teach me how to play bridge and to improve my chess skills with a lot of patience and very little success. Michael Elhadad and the members of my thesis defense committee proof-read parts of this thesis and gave very helpful suggestions.

I have spent great times with many other friends: Raffaele Giancarlo, Kunsoo Park, Xiangdong Yu, Roberto Grossi, Cosimo Spera, Dora Giammarresi, Moti Yung, Stuart Haber, Anat and Ido Ein-Dor, Bill Schillit, Jong Lim, George Michaels, Ohad Aloni, John Ioannidis, Frank, Yoelle and Uzi Maarek-Smadja, Yael and Adi Brender, Bülent Yener, Melik Isbara, Yaniv Hakohen and Orit Gwirceman.

Finally, I would like to thank Laura for being so patient and helpful while I was writing this thesis.

The IBM corp. awarded the fellowship that supported me for the past two years. The research reported in this thesis has been supported also by NSF Grants CCR-86-05353, CCR-88-14977 and CCR-90-14605.

To my parents, Erika and Robert
and my sister Iris

Chapter 1

Introduction

Problems involving strings arise in many areas of computer science and have numerous practical applications in text editing, information retrieval, lexical analysis, data compression, coding theory, formal languages and several other fields. In this dissertation we consider few problems involving strings from a theoretical perspective and provide efficient algorithms and lower bounds for these problems in sequential and parallel computation models.

We start with an introduction to the basic notation and the models of computation used and then present an overview of the problems considered and a summary of the new results.

1.1 Methods of Analysis

Although the problems we study have sometimes practical applications, we consider the analysis of the algorithms and lower bounds presented in this thesis from a theoretical perspective. We concentrate on the way the resource usage (computing time, memory space, number of processors) grows with increasing problem sizes. Therefore, we compare the efficiency of different algorithms based on formulae for their resource usage, rather than on experimental information. We also ignore for most part the constant factors in these formulae, giving them only up to orders of magnitude. That is, in most cases we concentrate on asymptotic growth rates of these formulae rather than on their exact mathematical expressions. The terminology used is standard and described next. For the motivation for this terminology and a more elaborate discussion see any algorithms textbook [3, 4].

We use the notation $O(f(n))$ ("big Oh" notation) to describe the set of functions that grow asymptotically *no faster* than $f(n)$. The notation $\Omega(f(n))$ describes the set of functions that grow asymptotically *at least* as fast as $f(n)$. Functions that grow asymptotically *at the same rate* as $f(n)$ are expressed by $\Theta(f(n))$. The formal definitions of O, Ω, and Θ are:

$g(n) \in O(f(n))$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) \leq cf(n)$, for all $n \geq n_0$

$g(n) \in \Omega(f(n))$ if there exists constants $n_0 \geq 0$ and $c > 0$ such that $g(n) \geq cf(n)$, for all $n \geq n_0$

$g(n) \in \Theta(f(n)$ if $g(n) \in \Omega(f(n))$ and $g(n) \in O(f(n))$.

Sometimes it is useful to say that a function has an asymptotic growth that is strictly smaller than another function. We denote by $o(f(n))$ the set of functions that grow strictly slower than $f(n)$ ("little oh" notation). The formal definition of $o(f(n))$ is:

$g(n) \in o(f(n))$ if and only if $\lim_{n\to\infty} \dfrac{g(n)}{f(n)} = 0$.

In this dissertation we consider only the *worst case* running times of our algorithms. That is, we consider those input instances of each size on which the algorithm takes the most time.

1.2 Properties of Strings

A *string* $\mathcal{S}[1..k]$ is a sequence of *symbols* $\mathcal{S}[i]$. The *length* of a string, k, is the number of symbols in the sequence. The set the symbols are chosen from is called *an alphabet* and will be denoted by Σ. We denote by $\Sigma_{\mathcal{S}}$ the set of symbols that are included in a string $\mathcal{S}$.

Many problems in computer science and other fields can be formalized in terms of strings. In fact, the whole theory of computation can be formalized in terms of machines that manipulate strings [57]. In this section we describe some basic properties of strings that will be used in the sequel.

1.2.1 The Alphabet

The assumptions we make on the alphabet Σ that the strings are chosen from has a crucial effect on the efficiency of the algorithms that can solve certain problems. We distinguish between four different types of assumptions that are made on the alphabet in the literature.

The most general assumption one can make on the alphabet is that any pair of symbols can be compared to obtain information about their equality. We distinguish between two classes of comparisons.

1. **A general alphabet** - Under this assumption any pair of symbols can be compared to obtain an equal or unequal answer. This type of comparisons are sufficient to solve almost all problems that will be discussed in this thesis [32, 48, 52, 66, 72].

2. **An ordered alphabet** - Under this assumption the alphabet is a totally ordered set and any pair of symbols can be compared to obtain a less than, equal, or greater than answer [7, 37, 38].

Some string problems require alphabet order in their definition but an arbitrary alphabet order can be helpful also for problems that do not require it. This is a very natural assumption since alphabet symbols are encoded numerically on a computer and this encoding introduces an order that can be used.

These two assumptions are often confused in the literature and are assumed to be the same. We stress that for some problems there is a significant difference in the possible solutions under these two different models.

A more restrictive assumption on the input alphabet is that the symbols of the alphabet are small integers. Again, we distinguish between two cases.

3. **A fixed alphabet** - Under this assumption the symbols of the alphabet are integers from a small range. This assumption is usually used in the literature to allow direct indexing of an array by an alphabet symbol [11, 40, 59, 62]

4. **A constant alphabet** - Under this assumption the symbols of the alphabet are integers from a constant range. As in the case of a fixed alphabet, input symbols can be used for direct indexing of an array of a constant size. This is especially useful for fast implementations of algorithms that use an automaton description or trie data structures [2, 5, 35, 60, 75, 89].

   In addition to the advantages of a fixed alphabet, this assumption allows several algorithms in the literature to pack few input symbols into a single register and compare them in a single operation [7, 46, 73]. This idea is usually referred to as the four-Russions technique after the Boolean matrix multiplication algorithm that was invented by Arlazarov, Dinic, Kronrod and Faradzev [3, 13].

To demonstrate the importance of the assumptions on the alphabet we consider a simple problem that has different solutions under these different assumptions.

The *element distinctness problem* is defined as follows: Given an input sequence $\mathcal{S}[1..k]$, check if all elements of the input sequence are distinct. That is, are there two different indices i and j such that $\mathcal{S}[i] = \mathcal{S}[j]$?

The following theorems characterize the complexity of this problem under the different assumptions on the set the elements of the sequence are chosen from.

Theorem 1.1 *If the only access an algorithm has to the input sequence is by comparisons of pairs of symbols that result in equal or unequal answers, then the upper and lower bounds for this problem are $\Theta(k^2)$.*

Proof: By comparing all $\binom{k}{2}$ pairs of elements, the element distinctness problem is obviously solved. That is, an $O(k^2)$ algorithm exists.

If an algorithm claims to solve the problem using fewer comparisons, an adversary can answer all the comparisons performed by the algorithm as unequal. If the algorithm concludes that all input symbols are distinct, since there is at least one pair of symbols that has not been compared by the algorithm and since inequality is not transitive, the adversary can define this pair of symbols to be equal. On the other hand, if the algorithm claims that two symbols are equal, the adversary can define all symbols to be different. Therefore, any algorithm that terminates without comparing all pairs of symbols can be fooled. □

An arbitrary order can help to solve the element distinctness problem as we show next.

Theorem 1.2 *The element distinctness problem can be solve in $O(k \log k)$ time by an algorithm that has access to the input sequence by comparing pairs of symbols and obtaining less than, equal or greater than answers.*

Proof: The elements of the input sequence can be sorted in $O(k \log k)$ time by a sorting algorithm such as merge-sort [3, 4]. In the sorted sequence, any elements that are equal appear consecutively and it suffices to compare each element to the next one to check if all elements are distinct. This results in an $O(k \log k)$ time algorithm. □

The algorithm described above is the best possible under the assumption that the only access an algorithm has to the input sequence is by comparisons as we show next. However, if we make stronger assumptions, the problem can be solved even more efficiently. This situation is identical to the classical example of sorting. An information theoretic lower bound implies that to sort a sequence of k elements $\Omega(k \log k)$ comparisons are necessary. However, if the sequence to be sorted contains only small integers, one can use bucket-sort that works in $O(k)$ time. For an introduction to sorting algorithms and the information theoretic lower bound see the chapters on sorting in [3, 4].

Theorem 1.3 *The element distinctness problem cannot be solved in fewer than $O(k \log k)$ order comparisons.*

Proof: We use the information theoretic lower bound for sorting. A proof of this bound can be found in standard textbooks [3]. That lower bound shows that a sequence of length k cannot be sorted in fewer than $\Omega(k \log k)$ comparisons and it holds even if all elements of the input sequence are distinct.

Assume an algorithm exists to solve the element distinctness problem in fewer than $\Omega(k \log k)$ order comparisons. The lower bound for sorting the same input sequence implies that by the information the algorithm has from the results of the comparisons it cannot know the exact order of all the elements in the sequence. This means that there is at least one pair of elements of the input sequence that have not been compared and their relative order is not implied by transitivity of the results of other comparisons.

An adversary has the choice of defining this pair of elements to be equal or to defined all elements to be distinct, fooling any algorithm that claims to solve the problem in fewer comparisons. □

Theorem 1.4 *If the elements of the input sequence are integers from the range $1..k$, then the element distinctness problem can be solved in $O(k)$ time.*

Proof: The sorting step in Theorem 1.2 can be implemented in $O(k)$ time using bucket-sort resulting in an $O(k)$ algorithm. A much simpler implementation without using bucket-sort is also possible. $\Box$

If the elements of the input sequence are chosen from a set of a constant size, an algorithm has even an additional advantage.

Theorem 1.5 *If the elements of the input sequence are integers from the range* $1..c$*, for a constant* c*, then the elements distinctness problem can be solved in constant time.*

Proof: If the sequence is long enough, that is, longer than c, the size of the set that the elements are chosen from, then it guaranteed by the pigeon-hole principle that there are two elements that are identical. Therefore, there is no need to check more than a constant number of elements of the input sequence. $\Box$

1.2.2 Periods

A string $\mathcal{S}[1..k]$ has *a period* length π if $\mathcal{S}[i] = \mathcal{S}[i+\pi]$ for $i = 1..k-\pi$. We define the set $\Pi^{\mathcal{S}[1..k]} = \{\pi_i^{\mathcal{S}} | 0 = \pi_0^{\mathcal{S}} < \pi_1^{\mathcal{S}} < \cdots < \pi_p^{\mathcal{S}} = k\}$ to be the set of all periods of $\mathcal{S}[1..k]$. $\pi_1^{\mathcal{S}}$, the smallest non-trivial period of $\mathcal{S}$ is called sometimes *the period* of $\mathcal{S}$.

A period π of a prefix of a string $\mathcal{S}[1..k]$ is called a *prefix period* of $\mathcal{S}[1..k]$. Let α be the largest integer smaller than or equal to k such that π is a period of $\mathcal{S}[1..\alpha]$. We say that the prefix period π *extends* to $\mathcal{S}[1..\alpha]$. If $\alpha < m$ we say that the prefix period π *terminates* at $\alpha + 1$.

The set of periods of a string has a special structure as shown by Lyndon and Schutzenberger [71]. Guibas and Odlyzko [55] and independently Colussi [31] gave a complete characterization of the possible structures this set can have. They have shown also that the set of periods is independent of the alphabet size. Their results are summarized in the following two lemmata.

Lemma 1.6 *(The periodicity lemma of Lyndon and Schutzenberger [71]) If a string* $\mathcal{S}[1..k]$ *has two periods* p *and* q *that satisfy* $p + q \leq k + gcd(p,q)$*, then* $\mathcal{S}$ *has also a period of length* $gcd(p,q)$*.*

Lemma 1.7 *(Characterization of all sets of periods of a string [31, 55]) Let* $\Psi = \{\psi_i | 0 = \psi_0 < \psi_1 < \cdots < \psi_z = k\}$ *and let* $\delta_i = \psi_{i+1} - \psi_i$*, for* $i = 0..z-1$*. The following statements are equivalent:*

1. *Ψ is a set of all periods of some string of length k over a binary alphabet.*
2. *Ψ is a set of all periods of some string of length k.*
3. *$\psi_i + \epsilon\delta_i \in \Psi$ for all integers ϵ such that $0 \leq \epsilon\delta_i \leq k - \psi_i$, and $k - \psi_{i+2} < \delta_i + gcd(\delta_i, \delta_{i+1})$.*

1.2.3 Local Periods

A *factor* or a *substring* of $\mathcal{S}[1..k]$ is a contiguous block of symbols $\mathcal{S}[n..m]$. A *factorization* of $\mathcal{S}[1..k]$ is a way to break $\mathcal{S}$ into several factors. In this section we consider factorizations of a string $\mathcal{S}[1..k]$ into two factors, $\mathcal{S}[1..l]$ and $[l+1..k]$. We call $S[1..l]$ a *left factor* or a *prefix* of $\mathcal{S}$ and $S[l+1..k]$ a *right factor* or a *suffix* of $\mathcal{S}$. Such a factorization is said to be *non-trivial* if none of the factors is equal to the empty string.

Given a factorization $(\mathcal{S}[1..l], \mathcal{S}[l+1..k])$ one can define *the local period* for the factorization as the shortest non-empty string that appears on both sides of the factorization. That is, the shortest string that matches $\mathcal{S}[1..l]$ aligned at the end of this string and also matches $\mathcal{S}[l+1..k]$ aligned at the start of this string. See Figure 1.1 for an example.

```
     a | b a a a b a
   b a   b a
```

(a)

```
       a b | a a a b a
   a a a b   a a a b
```

(b)

```
   a b a | a a b a
       a   a
```

(c)

Figure 1.1: The local periods at the first three non-trivial factorizations of '*abaaaba*'. Note that in some cases the local period can overflow to either side; this happens when the local period is longer than one of the two factors. The factorization in **(b)** is a critical factorization.

A non-trivial factorization of a string $\mathcal{S}$ is called a *critical factorization* if the local period at the factorization is of the same length as the period of $\mathcal{S}$. The following lemma shows that critical factorizations exist.

Theorem 1.8 *(The critical factorization theorem [28, 70]) Let $\mathcal{S}[1..k]$ be a string that has period length $\pi_1^{\mathcal{S}}$. Then, if we consider any $\pi_1^{\mathcal{S}} - 1$ consecutive non-trivial factorizations at least one is a critical factorization.*

1.2.4 Squares

A string $\mathcal{S}$ is called *primitive* if it is not a repetition of shorter string. That is, it cannot be written as $\mathcal{S} = w^k$, where w is a shorter string repeated k times.

A string of the form $w^2 = ww$, where w is a primitive non-empty string is called *a square*. A string that does not contain any square as a sub-string is called *square-free*. For example, the strings '*aa*', '*abab*' and '*baba*' are squares that 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 three letter alphabet that are square-free as shown by Axel Thue [84, 85] at the beginning of the century.

The maximal number of squares that a string of length k can have is $\Theta(k \log k)$ as shown by Crochemore [35] who gave the following construction:

Theorem 1.9 *(Crochemore [35]) Define the* Fibonacci strings *as* $f_0 =$ '*b*', $f_1 =$ '*a*' *and* $f_{q+1} = f_q f_{q-1}$. *Then for any integer* $q \geq 5$, *the number of squares in the* f_q *is larger than*[1] $\frac{1}{6}|f_q| \log |f_q|$.

1.2.5 Palindromes

A palindrome is a string that reads the same forward and backward. Formally, $\mathcal{S}[1..k]$ is a palindrome if $\mathcal{S}[i] = \mathcal{S}[k+1-i]$ for $i = 1..k$, or alternatively if $\mathcal{S} = \mathcal{S}^R$ and $\mathcal{S}^R$ is the string $\mathcal{S}$ reversed. A string $\mathcal{S}[1..n]$ has an *initial palindrome* of length k if the prefix $\mathcal{S}[1..k]$ is a palindrome. We make a small distinction between palindromes that have even or odd lengths.

Fact 1.10 *A string* $\mathcal{S}$ *of even length is a palindrome if and only if* $\mathcal{S} = ww^R$ *for some string* w. *A string* $\mathcal{S}$ *of odd length is a palindrome if and only if* $\mathcal{S} = waw^R$ *for some string* w *and some alphabet symbol* a.

Palindromes were known as word puzzles for hundreds of years. Some palindromes from Galil's [45] paper are listed below.

A man, a plan, a canal, Panama.
Dennis and Edna Sinned.
Red rum, sir, is murder.
Was it a rat I saw?
No lemons, no melon.

1.3 Computation Models

The choice of model of computation has an important role in the algorithmic techniques used and the results that can be obtained. The models that are used in this dissertation are standard models and their description can be found in several text books [3, 4, 54]. We present a short introduction to these models and their features that are significant for our work.

[1] All logarithms in this dissertation are to the base 2 unless specified otherwise.

1.3.1 The Random Access Machine

The Random Access Machine (RAM) is the standard model of computation used for describing sequential algorithms. It is very similar to most available commercial computers and algorithms described for this model can be implemented usually without much difficulty on any available computer. We will not give a formal description of this model here, but rather describe the features and limitations of this model that might not be very obvious. For a formal discussion see [3, 4].

As a theoretical model, the RAM has infinite amount of memory cells, each capable to store an integer of infinite size. We will limit this abstraction by assuming that only a finite amount of memory is used by our algorithms and that the size of the memory cells is limited.

As we will see later, the size of the memory cells can be crucial for some algorithms. The standard assumption will be that memory cells are large enough to hold the size of the input problem, but not much larger. By this we mean that if the size of the input problem is n, memory cells can hold integers which are bounded by a fixed degree polynomial in n.

The main affect that the latter restriction has on the algorithms overviewed in this thesis is that when the four-Russions technique is used, we can pack only $O(\log n)$ symbols into a single register.

1.3.2 The Parallel Random Access Machine

In this dissertation we consider only algorithms for a family of shared memory parallel computation models called the Parallel Random Access Machine (PRAM). We assume that the reader is familiar with the PRAM family of models and the standard algorithmic techniques in this model. For an introduction to these models and the common algorithmic techniques see [41, 54, 61]. We present next a brief overview of some essential facts.

The PRAM family of models consists of several synchronous processors that have access to a shared memory and each processor has the complete capabilities of a Random Access Machine. There are several versions of this model that differ in the ability of the processors to access the shared memory simultaneously.

- The weakest model in this family is the exclusive-read exclusive-write (EREW-PRAM) model where at each step simultaneous read operation and write operations at the same memory location are not allowed.

- A more powerful model is the concurrent-read exclusive-write (CREW-PRAM) model where only simultaneous read operations are allowed.

- The most powerful model in this family is the concurrent-read concurrent-write (CRCW-PRAM) model where read and write operations can be simultaneously executed.

In the case of the CRCW-PRAM model, there are several ways of how write conflicts are resolved.

- The weakest model, called the *common* CRCW-PRAM assumes that when several processors attempt to write to a certain memory location simultaneously, they all write the same value.
- A stronger model called the *arbitrary* CRCW-PRAM assumes that when several processors attempt to write to a certain memory location an arbitrary processor will succeed and the value it was trying to write will be stored at that memory cell.
- An even stronger model, the *priority* CRCW-PRAM assumes each processor has a preassigned priority and the highest priority processor succeeds to write.

These three versions of the CRCW-PRAM model differ in their ability to solve certain problems as shown by Boppana [19]. All the CRCW-PRAM algorithms described in this dissertation can be implemented in the common model. In fact these algorithms can be implemented even if we assume that the same constant value is always written in case of concurrent writes. However, to simplify the presentation we will sometimes use the more powerful priority model.

When studying algorithm in this family of computation models one has to consider few parameters. In addition to the time and space that are considered in sequential algorithms one has to account also for the number of processors deployed.

If an algorithm runs for time t on p processors we say that the total number of operations performed by the algorithm is the time-processor product tp. This is the potential number of operation that can be performed since some processors can be idle part of the time. We say that a parallel algorithm achieves *optimal speedup*, or that it is *optimal*, if the total number of operations performed by the algorithm is identical to that of the fastest sequential algorithm. Our goal will be to design fast parallel algorithm that achieve optimal speedup.

The following theorem will be used throughout the paper.

Theorem 1.11 *(Brent [23]) Any PRAM algorithm of time t that consists of x elementary operations can be implemented on p processors in $\lceil x/p \rceil + t$ time. (Elementary operations are the operations that are supported by each of the processors.)*

Brent's theorem has two aspects that are used in this dissertation:

1. Algorithms can be slowed down by trading processors for time. In particular a constant time p-processor algorithm can run in $O(t)$ time on $\lceil p/t \rceil$ processors.

2. Instead of counting the number of processors used by an algorithm we can count the total number of operations and analyze the running time. Brent's theorem gives the actual running time on any number of processors.

The following two lower bounds will be used in the sequel to argue that certain fast computations are not possible.

Theorem 1.12 *(Cook, Dwork, Reischuk [34]) Any CREW-PRAM algorithm that computes the* Boolean AND *function of n input bits takes $\Omega(\log n)$ time on any number of processors.*

Theorem 1.13 *(Beame and Hastad [14]) A CRCW-PRAM algorithm that computes the* parity *function of n input bits takes $\Omega(\frac{\log n}{\log\log n})$ time on any polynomial number of processors.*

The next theorem is used extensively in all the parallel algorithms that we describe.

Theorem 1.14 *(Fich, Ragde and Wigderson [42]) The minimum of n integers in the range $1..n$ can be computed by an n-processor CRCW-PRAM in constant time.*

1.3.3 The Comparison Model

The comparison model is an unrealistic model of computation that accounts only for comparisons and ignores computation and communication between processors.

The comparison model can be described as a decision tree. Each node of the tree contains a list of comparisons to be performed and the outgoing branches from a node lead the next nodes of comparisons to be performed. The computation starts from the root of the tree and proceeds according to the outcome of the comparisons. When a leaf of the tree is reached it is assumed that the problem is solved.

We can distinguish between sequential and parallel models. In a sequential model each node contains a single comparison between a pair of symbols and two or three[2] outgoing branches. In the parallel model each node contains several comparisons that are performed simultaneously and there are several outgoing edges. The number of comparisons performed at each node is an analog to the number of processors available on a PRAM.

There are two reasons that we use the comparison model in this thesis:

- Lower bounds on the comparison model, that is lower bounds on the minimal depth of a decision tree that solves a problem apply also as lower bounds for the RAM and PRAM models when the algorithms have access to the input strings only by comparisons of alphabet symbols. The lower bounds apply in this case because each step of computation on a RAM or a PRAM is either a comparison or other computation. Bounding the number of comparisons steps obviously provides a lower bound on the total running time of an algorithm.

- It is possible to exactly count the number of comparisons performed by a comparison model algorithm and obtain bounds on the complexity without ignoring constant factors as we do in the more standard models.

[2]Depending on the type of comparisons used; i.e. order comparisons or equality comparisons.

The relation between the comparison model and the RAM or PRAM models can be also misleading sometimes.

- The other computation except comparisons on RAM or PRAM is sometimes expensive and should not be ignored. This means that it is possible to obtain efficient comparisons model algorithms that cannot be realized efficiently on a standard model. These algorithms can still be interesting from a theoretical perspective.

  An extreme example of this is sorting in parallel. On the comparison model a sequence of n elements can be sorted in a single step that compares all pairs of elements. On the other hand, Theorem 1.13 implies that such a sequence cannot be sorted in less than $\frac{\log n}{\log \log n}$ time with any polynomial[3] number of processors. An intermediate compromise that is called chain sorting [56] can be implemented more efficiently on a PRAM.

- Lower bounds on the comparison model do not hold for RAM or PRAM algorithms that can access the input by operations other than comparisons. Such algorithms work under the assumption of a fixed or a constant alphabet and can beat the lower bounds for the comparison model. This is exemplified by the fact that bucket-sort is faster than the information theoretic lower bound for sorting.

1.4 Summary of Results

We consider the following problems in this dissertation:

- String matching – Given two input strings $\mathcal{P}[1..m]$ and $\mathcal{T}[1..n]$ called a pattern and a text, respectively, find all occurrences of the pattern string in the text string.

  The string matching problem is one of the simplest pattern matching problems and it occurs naturally as part of data processing, data compression, text editing, lexical analysis and information retrieval. Techniques for solving string matching problem usually have immediate implications to other string problems.

  Research on string matching and other string problems has been stimulated recently by a growing interest from various other fields. In particular, molecular biologist are faced today with problems of searching for regularities and similarities in hugh data-bases of DNA and protein sequences[4]. These data-bases are growing very fast and provide new challanges to computer scientists. The need to solve these hugh problems justifies the use of more complicated algorithms that were not practical for smaller problems.

---

[3] A polynomial in n.

[4] DNA and proteins are built from a small number of types of building blocks and can be modelled as sequences of this blocks. The problems that need to be solved in practice are generalized matching problems where some changes or mutations are allowed.

- Finding all periods of a string – Given an input string $\mathcal{S}[1..n]$ find all periods of $\mathcal{S}$.

  Periods and periodicity properties underlay most efficient string matching algorithms. The study of periodicity properties helps in the development of the string algorithms and the lower bounds presented. The algorithm we develop to compute all periods of a string applies immediately also to find all initial palindromes.

- Testing for squares – Given an input string $\mathcal{S}[1..n]$, test if it is square-free and report a square if one is found.

  Squares and strings that contain squares have important implications in coding theory, automata theory, formal language theory, data compression and recently also in molecular biology. Squares are used implicitly in many string algorithm since the basic periodicity lemma (Lemma 1.6) can be easily applied when a substring is repeated at least twice.

- Finding all initial palindromes – Given an input string $\mathcal{S}[1..n]$ find all the initial palindromes it contains.

  Palindromes have been studied for centuries as word puzzles and have some uses in formal language theory. Although the definitions of squares and palindromes are very similar, there is a big computational difference between these two terms. Manacher [74] gave a simple and elegant linear-time algorithm that can find all palindromes in a string. However, $O(n \log n)$ time is necessary even to test if a string over a general alphabet contains a square, as shown by Main and Lorentz [72]. An other example comes from formal language theory: the language of all strings that contain palindromes over a given alphabet is easily shown to be *context-free* [57], while the language of all strings that contain squares is not *context-free* as shown by Ross and Winklmann [79].

In the sequential setting, we study the exact number of comparisons that need to be performed by any comparison based string matching algorithm. The following result is presented:

- A new algorithm that performs better for longer patterns. The new algorithm performs $n+\frac{c \log m}{m}n$ comparisons in the worst case showing that, in general, searching for longer patterns is easier than searching for short patterns.

In the parallel setting, we study the complexity of several problems and provide tight upper and lower bounds for:

- Parallel string matching.

- Finding all periods of a string.

- Detecting squares in a string.

- Finding all initial palindromes of a string.

All these problems can be trivially solved in constant time by a CRCW-PRAM with a polynomial number of processors (a small degree polynomial). We present optimal speed-up parallel CRCW-PRAM algorithms and give tight bounds for any number of processors used.

Chapter 2

The Exact Complexity of String Matching

String matching is the problem of finding all occurrences of a short string $\mathcal{P}[1..m]$ called a *pattern* in a longer string $\mathcal{P}[1..n]$ called a *text*. While the problem is very easily stated and many of the simple algorithms perform very well in practice, numerous works have been published on the subject and research is still very active. The string matching problem is one of the simplest pattern matching problems and it occurs naturally as part of data processing, data compression, text editing, lexical analysis and information retrieval. Techniques for solving string matching problem usually have immediate implications to other string problems.

Research on string matching and other string problems has been stimulated recently by a growing interest from various other fields. In particular, molecular biologist are faced today with problems of searching for regularities and similarities in hugh data-bases of DNA and protein sequences These data-bases are growing very fast and provide new challanges to computer scientists. The need to solve these hugh problems justifies the use of more complicated algorithms that were not practical for smaller problems.

A naive algorithm that solves the string matching problem can proceed as follows: Consider the first $n - m + 1$ positions of the text string. Occurrences of the pattern can start only at these positions. The algorithm checks each of these positions for an occurrence of the pattern. Since it can take up to m comparisons to verify that there is actually an occurrence of the pattern at a given text position, the time complexity of this algorithm is $O(nm)$. Note that the only operations involving the input strings in this algorithm are comparisons of pairs of symbols.

Indeed, the only assumption one needs to solve the string matching problem is that symbols of the input strings can be compared to obtain information about their equality. Most known string matching algorithms use only this information. However, some algorithms make further assumptions on the inputs strings.

In this chapter we study the number of comparisons performed by string matching algorithms that have access to the input strings only by comparing pairs of symbols to test whether they are equal or not. We are interested in the exact number of comparisons performed rather than in an asymptotic bound. The study of the exact number of comparisons necessary to solve a problem is not new. There is extensive literature on several order statistics problems. It is easy to show that $n-1$ comparisons are necessary and sufficient to find the minimum or the maximum of n elements. It has been shown also that computing both the minimum and maximum requires $\lceil \frac{3}{2}n - 2 \rceil$ comparisons [76] and that computing the largest two elements requires $n-1+\lceil \log n \rceil$ comparisons [65, 82]. The exact complexity of finding the third largest element is known for all but a finite number of cases [63, 64, 90]. The complexity of sorting is determined up to sorting 12 elements [3, 44]. The exact complexity of finding the median is another intriguing open problem: it has been known for several years that the number of comparisons required is between $2n$ [15] and $3n$ [81].

In this chapter we present a new string matching algorithm with improved bounds on the number of comparisons performed. The new algorithm requires at most $n + o_{m\to\infty}(1)n$ comparisons, closing the gap between the lower and upper bounds as the length of the pattern grows [26].

2.1 Historical Overview

There are several algorithms that solve the string matching problem in linear time. For a survey on string matching algorithms see Aho's [1] paper. Most of these algorithms work in two phases: in the first phase the pattern is preprocessed and some information is computed to be used later in a text processing phase that finds the actual occurrences of the pattern in the text.

The Knuth-Morris-Pratt [66] algorithm is the first linear time that was discovered. It dates back to *1969* when Morris implemented an early version of the algorithm in a text-editor for the *CDC 6400* computer. The discovery of the algorithm has a very interesting history as reported in the paper [66] that was published only in *1977*. Most notable probably is the fact that a result of Cook [33] on linear-time simulation of two-way deterministic pushdown automaton by a random access machine implied that a linear-time string matching algorithm exists. Knuth went through the laborious details of Cook's construction and discovered the algorithm. The algorithm reported in the final paper is a refined version of these two independent discoveries.

The Knuth-Morris-Pratt [66] algorithm performs at most $2n$ comparisons in its text processing phase. The Boyer-Moore algorithm performs $3n$ comparisons in the worst case as proven recently by Cole [29], but seems to perform better in practice. A variant the Boyer-Moore algorithm that performs $2n$ comparisons in the worst case was designed by Apostolico and Giancarlo [10].

All these algorithms use an $O(m)$ auxiliary space. At a certain time it was known that a logarithmic space solution was possible [51], and the string matching problem was conjectured to have a time-space trade-off [21]. This conjecture was later disproved when a linear-time constant-space algorithm was discovered by Galil and Seiferas [38, 52]. It was shown that even a *6-head two-way* finite automaton can perform string matching in linear time. It is still an open problem whether a k-head one-way finite automaton can perform string matching. The only known cases are for $k = 1, 2, 3$ [53, 68, 69] where the answer is negative.

The Aho-Corasick [2] algorithm which can search for multiple patterns in linear time is also used widely in practice. This algorithm works under the constant alphabet assumption and it does not use character comparisons. It uses direct indexing to an array and therefor it does not fall in the class of algorithms studied in this chapter.

A recent work by Colussi [32] showed that the string matching problem can be solved using $\frac{3}{2}n$ comparisons improving the previous bound of $2n$ that was achieved by several algorithms [22, 38, 66]. Colussi's bound was later improved to $\frac{4}{3}n$ by Galil and Giancarlo [48] who proved also lower bounds that match their upper bound for short patterns [49]. These lower bounds were generalized and improved by Zwick and Paterson [91] who presented also other results that we will mention later.

In this chapter we concentrate on the complexity of the string matching problem in a deterministic comparison model that can perform comparisons to test if two symbols are equal or not. In the special case of a binary alphabet the string matching problem can be solved in n comparisons since inequality implies equality to the other symbol. Therefore, we will be interested in solving the problem when the size of the alphabet is larger.

To eliminate the possibility of obtaining equality information from inequality answers it suffices to assume that the text string can have an extra symbol that does not appear anywhere in the pattern string. In this case, an efficient algorithm should never compare any text symbol to this extra symbol since an adversary can always answer such comparisons as unequal without giving the algorithm any additional information. If an algorithm never compares a text symbol to this extra symbol, the only equality from inequality information an algorithm can get is by comparing a symbol to all pattern symbols and getting negative answers, but in this case it already knows that this particular text symbol does not appear anywhere in the pattern.

In the comparison model we count only the number of comparisons performed and the actual running time of an algorithm on a standard model is of secondary interest. It might be impossible to match the number of comparisons performed by a comparison model algorithm with an algorithm that works in linear time on a conventional model. For this reason, only the number of comparisons performed by the algorithms described in this chapter are analyzed and efficient implementations are not discussed.

All previous algorithms that solve the string matching problem are oblivious in the sense that they sometimes "forget" or do not use information that was obtained in previous comparisons. Colussi [32] developed his algorithm by using program correctness proof techniques to avoid performing comparisons which are not necessary as a result of previous comparisons. The algorithms described in Section 2.2 do not forget answers to previous comparisons, which makes their analysis much easier.

To overcome the obliviousness problem and try to find an algorithm that performs minimal number of comparisons we suggest the following idea, which we call a *buffer* (this idea is not used by the algorithms presented in this chapter), that can help to get better bounds for algorithms that "forget": The buffer, which is placed between the algorithm and the input strings, will answer any comparisons that are implied by the information known so far without charging the algorithm for these comparisons. Such a buffer can be implemented efficiently in a standard model, in some cases even in linear time, but is not practical since usually it is easier to make the actual comparisons. However, some versions of our algorithms which are described in Sections 2.2.5 and 2.2.7 are almost identical to Colussi's [32] and Galil and Giancarlo's [48] algorithms if they are modified to use a buffer.

The length of the text, n, can be much larger than the length of the pattern, m. To simplify the analysis, we assume that any pattern preprocessing is free and we do not account for it in our bounds. Note that an algorithm can obtain complete information about the pattern in $\frac{m(m-1)}{2}$ comparisons that will not be accounted for. If order comparisons are allowed, then $O(m \log m)$ comparisons suffice to get the same information. The preprocessing step of all algorithms mentioned above can be implemented in linear time.

There is also another reason for not accounting for the preprocessing. All the lower bounds for this problem in the literature involve a fixed pattern string. Since we try to get matching lower and upper bounds, a lower bound in the comparison model has the advantage of the pattern string being fixed and not having to account for preprocessing it.

Galil and Giancarlo [48] distinguish between on-line and off-line algorithm. An on-line algorithm has access to the text through a sliding window whose length is equal to the length of the pattern and it has to report if there is an occurrence at a certain text position i before examining any text position larger than or equal to $i+m$. In contrast, an off-line algorithm has access to the whole text. Note that in a sliding window algorithm the length of the window must be larger than or equal to the length of the pattern; an off-line algorithm can be thought of as an algorithm with a sliding window of infinite length. We follow the notation of Zwick and Paterson [91] and say that an algorithm with a sliding window of length $m+k$ has k look-ahead. For the sake of consistency we will call a zero-look-ahead algorithm *on-line* and an infinite-look-ahead algorithm *off-line*.

We define the function $c^l_{\mathcal{P}[1..m]}(n)$ to denote the number of comparisons necessary to find all occurrences of a particular pattern $\mathcal{P}[1..m]$ in a text string of length n for a l-look-ahead algorithm. By the notation described above we define $c^{on-line}_{\mathcal{P}}(n) = c^0_{\mathcal{P}}(n)$ and $c^{off-line}_{\mathcal{P}}(n) = c^{\infty}_{\mathcal{P}}(n)$. Obviously, if $k \leq l$ then $c^l_{\mathcal{P}}(n) \leq c^k_{\mathcal{P}}(n)$, and $c^{off-line}_{\mathcal{P}}(n) \leq c^{on-line}_{\mathcal{P}}(n)$. We define also $c^l(n,m)$ to be the maximum of $c^l_{\mathcal{P}}(n)$ over all pattern strings $\mathcal{P}$ of length m.

Colussi [32] showed that $c^{on-line}(n,m) \leq 1.5n - .5(m-1)$ while the best previous bound was $c^{on-line}(n,m) \leq 2n - m$ [10, 38, 66]. Galil and Giancarlo [48] proved that the number of comparisons performed by Colussi's algorithm can be bounded also by $n + \lfloor (n-m)\frac{\min(\pi_1, m-\pi_1)}{m} \rfloor \leq n + \lfloor \frac{n-m}{2} \rfloor$, where π_1 is the length of the shortest period of the pattern. This bound is achieved also by an algorithm of Apostolico and Breslauer [8], that is obtained by a combination of the techniques of Knuth, Morris and Pratt [66] and Crochemore and Perrin [38], but is much easier to analyze. Galil and Giancarlo

[48] presented also an improved version of Colussi's algorithm that performs $n + \lfloor (n - m)\min(1/3, \frac{\min(\pi_1, m-\pi_1)+2}{2m})\rfloor \leq \frac{4}{3}n - \frac{1}{3}m$ comparisons in the worst case.

In another paper, Galil and Giancarlo [49] prove that if m is an odd integer that is larger than or equal to three, then for infinitely many n's

$$c^{on-line}(n,m) \geq n + \lfloor \frac{2(n-m)}{m+3} \rfloor$$

and if m is an even integer that is larger than three, then for infinitely many n's

$$c^{on-line}(n,m) \geq n + \lfloor \frac{2(n-m)}{m+4} \rfloor.$$

For patterns of length 3, this bound matches Galil and Giancarlo's upper bound, determining that $c^{on-line}(n,3) = \frac{4}{3}n - \frac{1}{3}m$. The cases where $m = 1, 2$ are not interesting since it is trivial to show that in this cases $c^{on-line}(n,m) = n$. Other lower bounds for off-line algorithms and for on-line algorithms that search only for the first occurrence of the pattern in the text are also presented in the same paper.

Zwick and Paterson [91] generalized the lower bound of Galil and Giancarlo [49] for on-line algorithms to

$$c^{on-line}(n,m) \geq n + \lfloor \frac{n-m}{\pi_2} \rfloor$$

where π_2 is the second period of the pattern if it exists, and $\pi_2 = \infty$ if the second period does not exist. This is a generalization of the Galil-Giancarlo [49] bound since it gives the same lower bounds for the pattern strings '$a^k b a^k$' and '$a^k b a^{k+1}$' used by Galil and Giancarlo for their bounds. Zwick and Paterson present also a similar bound for off-line algorithms and show that an off-line algorithm that is not allowed to compare pairs of text symbols must perform $\frac{4}{3}n$ comparisons to find all occurrences of 'aba', the same number of comparisons that an on-line algorithm must perform for this pattern.

In the same paper, Zwick and Paterson show also that a look-ahead is useful, at least in some cases. They show that an on-line algorithm must take $\frac{5}{4}n$ comparisons to find all occurrences of '$abaa$' in a text of length n (this bound is actually achieved by the Galil-Giancarlo [48] algorithm); while a 4-look-ahead algorithm can find all occurrences of the same pattern in $\frac{8}{7}n$ comparisons, and this number of comparisons is necessary even for an off-line algorithm.

2.2 A Fast Algorithm

In this section we study the exact number of comparisons necessary to solve the string matching problem and show that searching for a long pattern is easier than searching for a short one. Our approach allows us to simplify the analysis of previous algorithms and obtain better bounds. In particular, we present an algorithm that finds all occurrences of a pattern of length m in a text of length n in $n + \frac{c \log m}{m}(n-m)$ comparisons.

2.2.1 The Structure of the New Algorithm

All the algorithms described in Section 2.2 are on-line and share the structure outlined in this section. This structure is described in an inefficient manner, and the emphasis is on the number of comparisons performed. Although our algorithm is developed in a different way from Colussi's [32] algorithm we obtain a very similar algorithm. The similarity to Colussi's algorithm will be discussed in Section 2.2.5.

Two important properties of the new algorithms that make them comparison efficient and easier to analyze are:

1. They do not "forget" any comparison.

2. The comparisons are accounted for during the computation.

In the next sections we will describe several versions of our algorithm and analyze the number of comparisons performed by each version. Each comparison is charged either to a text position or to a special fund. The number of comparisons charged to the special fund has to be analyzed separately for each version of the algorithm, while the number of comparisons that are charged to text positions is smaller then n and will be analyzed in this section.

The new algorithm proceeds by comparing text symbols from left to right in a "forward step" leaving some "holes", which are text positions that are not known. These holes will be "filled" or compared to later if necessary in a "backward step" that compares symbols from right to left.

We maintain an integer ζ that is the next text position that the algorithm considers and a set $\Phi^\zeta = \{\phi_i^\zeta | \zeta - m < \phi_1^\zeta < \phi_2^\zeta < \cdots < \phi_{q_\zeta}^\zeta \leq \zeta\}$ of all positions of the text up to ζ that can still start occurrences of the pattern and have not been reported as occurrences yet. Each member of Φ^ζ may have a single credit assigned to it or no credit at all. The following invariants will be maintained:

1. The set Φ^ζ contains exactly all indices in the text, up to ζ, that are possible occurrences of the pattern by answers to comparisons so far.

2. If we align copies of the pattern starting at all text positions in Φ^ζ and look at the columns under text positions $\phi_1^\zeta \cdots \zeta$, the symbols in all copies of the pattern at each of these columns are identical.

Note that invariant one does not imply invariant two since some comparisons are skipped.

The third invariant concerns credits. Initially, each text positions has a single credit which can be used for a single symbol comparison. Each credit can be assigned and later reassigned to a member of Φ^ζ.

3. The credit of a text position can be assigned or reassigned only to members of Φ^ζ that are larger than or equal to the text position that the credit belonged to originally.

We maintain this invariant by reassigning credits only to larger members of Φ^ζ; i.e. credits in Φ^ζ can only move to the right. In addition, the members of Φ^ζ without credits are always the first ones.

The algorithm starts with $\zeta = 1$ and $\Phi^0 = \emptyset$. We describe how the set Φ is updated when we advance from text position $\zeta - 1$ to the next text position ζ. The set Φ^ζ is initially constructed by adding ζ to $\Phi^{\zeta-1}$ and then it may undergo modifications. The modifications performed are only deletions of members of Φ^ζ until the invariants are satisfied. In the description below we assume that each time some members are deleted from Φ^ζ the remaining members are renumbered and called $\phi_1^\zeta \cdots \phi_{q_\zeta}^\zeta$.

The initial value of the set Φ^ζ obviously satisfies invariant one which will be maintained by removing from the set Φ^ζ only members for which there is evidence that they can not start an occurrence of the pattern. However, invariant two might be violated. We repeat the following until invariant two is satisfied. We distinguish between two cases:

- Invariant two is satisfied. This means that if we align copies of the pattern starting at all text positions in Φ^ζ and look at the column under text position ζ, the symbols in all copies of the pattern at that column are identical.

  If $\zeta \in \Phi^\zeta$, the credit of text position ζ is assigned to $\phi_{q_\zeta}^\zeta = \zeta$ in the set Φ^ζ satisfying invariant three.

  Note that no comparison is made. We call this text position ζ a *hole* and record it for a later processing. See Figure 2.1 for an example.

  The similarity to the holes in the algorithms of Colussi [32] and Galil and Giancarlo [48] is discussed in Sections 2.2.5 and 2.2.7.

- Invariant two is violated. This means that if we align copies of the pattern starting at all text positions in Φ^ζ and look at the column under text position ζ, then some symbols are different.

  Comparisons are performed between text position ζ and the symbol that is under ζ in a copy of the pattern that is aligned starting at one of the text positions in Φ^ζ and the set Φ^ζ is modified according to the outcome of the comparisons, until invariant two is satisfied. This loop will eventually terminate since at each step some members of Φ^ζ are removed.

  If all members of Φ^ζ except ζ have credits assigned to them, we compare text position ζ with the symbol under this position in the first copy of the pattern aligned starting at ϕ_1^ζ. If some members of Φ^ζ except ζ do not have credits assigned to them, the symbol we compare to text position ζ depends on the version of our algorithm and will be discussed later.

  - The comparison results in an equal answer.

    We remove from Φ^ζ all text positions that if we align a copy of the pattern starting at these positions the symbol under text position ζ is not the same as the symbol in text position ζ. Invariants one and two are satisfied and the construction of Φ^ζ is completed. The comparison is charged to the credit of text position ζ. This text position will never be compared again since we know which symbol of the pattern is there.

Since at least one of the members of Φ^ζ was eliminated, if any of the eliminated members had a credit and if $\zeta \in \Phi^\zeta$, that credit can be reassigned to $\phi^\zeta_{q_\zeta} = \zeta$. Note that in this case the credit is reassigned to a larger position without violating invariant three. If no credit is assigned to $\zeta \in \Phi^\zeta$, all deleted members of Φ^ζ had no credit and at least one was deleted. In any case the number of members of Φ^ζ that have no credit is not larger than the number of members without credit that the set Φ^ζ had before this comparison was made.

We now have to shift credits to the right to maintain invariant three.

– The comparison results in an unequal answer.

We remove from Φ^ζ only the text positions such that if we align a copy of the pattern starting at these positions the symbol under text position ζ is the same as the symbol that we compared to text position ζ. These are the only text positions in Φ^ζ for which we have evidence of not starting an occurrence of the pattern.

If all members of Φ^ζ except ζ had a credit we have just compared text position ζ to the symbol under text position ζ in the copy of the pattern aligned at ϕ^ζ_1. Since the comparison failed, ϕ^ζ_1 was eliminated from Φ^ζ and the comparison is charged to the credit ϕ^ζ_1 had.

If some members of Φ^ζ except ζ did not have a credit we have just compared text position ζ to some symbol depending on the version of our algorithm we describe later. In this case the comparison is charged to the special fund.

The total number of comparisons charged to the special fund depends on the version of the algorithm we use and will be analyzed separately for each version.

We now must go back to check if invariant two is satisfied.

If the set Φ^ζ is not empty after its construction has been completed, we must check if $\phi^\zeta_1 + m = \zeta + 1$ since our algorithm is on-line and it cannot examine a text position greater than ζ before reporting if there is an occurrence starting at text position ϕ^ζ_1.

In this case we must check if there is actually an occurrence of the pattern starting at text position ϕ^ζ_1 and remove ϕ^ζ_1 from Φ^ζ. To verify that there is such an occurrence we must go back to fill the holes we left while constructing the set Φ^ζ. We scan the holes that are in text positions greater than or equal to ϕ^ζ_1 in decreasing order and compare the symbol at the text positions of the holes to the corresponding symbol of a copy of the pattern aligned at text position ϕ^ζ_1 until a mismatch is found or all holes are exhausted. Once a text position of a hole is compared it is removed from the list of holes and will not be considered a hole any longer.

The comparisons performed are charged to the initial credit that the text position at each hole had. By invariant three this credit could be reassigned only to a member of Φ^ζ

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	ζ									
The text	**a**	**b**	**a**	**a**	b	**a**	b	**a**	**b**	**a**	.	.	.					
ϕ_1^ζ			a	a	b	a	b	a	a	b	a	a						
ϕ_2^ζ								a	a	b	a	b	a	a	b	a	a	
ϕ_3^ζ									a	a	b	a	b	a	a	b	a	a

Figure 2.1: The state of the algorithm at text position $\zeta = 9$. Holes are listed in boldface. Note that there is a hole at text position 9 since symbols at this column in the copies of the pattern aligned starting at text positions in $\Phi^\zeta = \{3, 8, 9\}$ are all the same. This means that no comparison is made at this text position and Φ^{10} will initially inherit all the text positions in Φ^9. Later, a comparison must be performed at text position $\zeta + 1 = 10$ because the column under this position contains different symbols. Since all text positions in Φ^{10} have credits the symbol at text position 10 will be compared to '*b*', the symbol aligned with it in the copy of the pattern that is aligned starting at text position 3, which is the smallest in Φ^{10}. The comparison fails, $\Phi^{10} = \{9, 10\}$, and a hole is left at text position 10. If we compared to '*a*' instead, the comparison would have succeeded and we would have got the same set Φ^{10}, but we would not have a hole at text position 10.

that is greater than or equal to the text position of the hole. This credit has not been used yet since credits of members of Φ are used only when ϕ_1, the smallest member of Φ is removed from Φ and the original text position the credit belonged to will never be examined later. Credits that were assigned to members of Φ that were removed at some point and were not reassigned could not have been used either. These credits may also pay for comparing holes.

After this step is completed the list of holes is emptied and all members of Φ^ζ lose their credits. This is the only way that additional members without credit are introduced into the set Φ.

In case of a mismatch all members of Φ^ζ that are smaller than or equal to the position of the mismatch are removed from Φ^ζ. This must be done to satisfy invariant one because all copies of the pattern aligned at these text positions have the same symbol in the column of the mismatch by invariant two.

If the comparisons in all the holes succeed an occurrence of the pattern can be reported at text position ϕ_1^ζ. In any case the algorithm resumes with the new set Φ^ζ, and all members of Φ^ζ do not have credits. The skeleton of the algorithm is summarized in Figure 2.2.

Theorem 2.1 *The algorithm correctly finds and reports all occurrences of the pattern in the text. All comparisons are either charged to a text position or to the special fund.*

– We use $\mathcal{P}[1..m]$ to denote the pattern and $\mathcal{T}[1..n]$ to denote the text.
$\Phi^0 = \emptyset$
$\zeta = 1$
repeat
 $\Phi^\zeta = \Phi^{\zeta-1} \cup \{\zeta\}$
 if $\phi_1^\zeta + m > n + 1$ **then**
  STOP – No need to proceed if the text is not long enough to include more occurrences.
 while there are two members $\phi_i^\zeta, \phi_j^\zeta \in \Phi^\zeta$ such that $\mathcal{P}[\zeta - \phi_i^\zeta + 1] \neq \mathcal{P}[\zeta - \phi_j^\zeta + 1]$ **do**
  If all members of Φ^ζ, except ζ, have credits assigned to them, choose $\delta = 1$.
   Otherwise choose δ according to the version of the algorithm we are using.
  if $\mathcal{P}[\zeta - \phi_\delta^\zeta + 1] = \mathcal{T}[\zeta]$ **then** – This is an actual comparison.
   Charge the last comparison to text position ζ.
   Remove all ϕ_ϵ^ζ's such that $\mathcal{P}[\zeta - \phi_\epsilon^\zeta + 1] \neq \mathcal{T}[\zeta]$ from Φ^ζ.
  else
   If ϕ_1^ζ has credit assigned to it, the last comparison is charged to its credit.
    Otherwise the comparison is charged to the special fund.
   Remove ϕ_δ^ζ and all ϕ_ϵ^ζ's such that $\mathcal{P}[\zeta - \phi_\epsilon^\zeta + 1] = \mathcal{P}[\zeta - \phi_\delta^\zeta + 1]$ from Φ^ζ.
  end
 end
 If no successful comparison was made at this step of the loop,
  then mark text position ζ as a hole.
 If $\zeta \in \Phi^\zeta$, then we need to assign a credit to ζ.
  If ζ is a hole it gets the initial credit of the text position.
  Otherwise, at least one member of Φ^ζ was removed.
   If a member that was removed had a credit, ζ inherits its credit.
   If all members that were removed did not have a credit and some members of Φ^ζ
    have credits, the credit of the smallest member of Φ^ζ that has a credit is
    reassigned to ζ.
 – The number of members without credit was not increased.
 if $\phi_1^\zeta + m = \zeta + 1$ **then**
  Let h_i be the indices of all holes at text positions greater than or equal to ϕ_1^ζ.
  scan h_i **in decreasing order**
   if $\mathcal{P}[h_i - \phi_1^\zeta + 1] \neq \mathcal{T}[h_i]$ **then** – This is an actual comparison.
    Remove all $\phi_\epsilon^\zeta \leq h_i$ from Φ^ζ.
    Stop scanning holes.
     – There is no occurrence of the pattern starting at text position ϕ_1^ζ.
   end
  end
  Unmark all holes.
  If the comparisons at all holes succeeded, report an occurrence
   of the pattern starting at text position ϕ_1^ζ and remove ϕ_1^ζ from Φ^ζ.
 end
 $\zeta = \zeta + 1$
end

Figure 2.2: The skeleton of the new algorithms.

At most one comparison is charged to every text position. The number of comparisons charged to the special fund depends on the version of the algorithm and will be determined later.

Proof: If there is a occurrence of the pattern starting at a text position ω, after $\Phi^{\omega+m-1}$ has been constructed it must include $\phi_1^{\omega+m-1} = \omega$ by invariant one. At this point holes are filled. Since an actual occurrence of the pattern starts at text position ω all comparisons must succeed and an occurrence will be reported. It is easy to see that non-existing occurrences cannot be reported as occurrences by the algorithm.

A comparison is charged to a text position directly only when it is successful or in a backward step. If the comparison was successful the algorithm knows what the text symbol at that position is and it will not compare that symbol again. A comparison that is unsuccessful in a backward step can also be charged to the text position compared since the set Φ^ζ is shifted ahead over the position of the mismatch and that text position will never be examined again. Other comparisons that are unsuccessful are charged either to a member of Φ or to the special fund. A comparison is charged to a member of Φ only if it is the first member and it had a credit. In this case this member will be removed from Φ and since the credit originally belonged to a text position smaller than or equal to this member, we are guaranteed that it will never be used again. $\Box$

2.2.2 Periodicity and the New Algorithm

There is a strong relation between the set Φ^ζ defined in the previous section and the sets of periods of prefixes of the pattern as we show in the following lemma. This relation means that most information that is required at each step of the new algorithm can be precomputed in a pattern preprocessing phase.

Lemma 2.2 *If the set Φ^ζ is not empty after its construction is completed, then it is equal to the set of all periods of the prefix of the pattern $\mathcal{P}[1..\zeta - \phi_1^\zeta + 1]$, except $\zeta - \phi_1^\zeta + 1$, shifted by ϕ_1^ζ. That is $\Phi^\zeta \cup \{\zeta + 1\} = \{\pi + \phi_1^\zeta | \pi \in \Pi^{\mathcal{P}[1..\zeta-\phi_1^\zeta+1]}\}$.*

Proof: Assume the set Φ^ζ is not empty. By invariant one, it must contain all text positions, up to ζ, that can still start occurrences of the pattern. Furthermore, any text position larger than ζ has not yet been examined.

If ϕ_1^ζ can start an occurrence of the pattern, then all text positions in the set $\{\pi + \phi_1^\zeta | \pi \in \Pi^{\mathcal{P}[1..\zeta-\phi_1^\zeta+1]} \; and \, \pi \neq \zeta - \phi_1^\zeta + 1\}$ can also start occurrences since if we aligned copies of the pattern at these text positions the overlapping parts of these copies with the first copy aligned at ϕ_1^ζ are identical up to text position ζ. This means that Φ^ζ must include all these text positions.

By invariant two if we aligned copies of the pattern at all text positions in Φ^ζ the symbols at each column starting from the column under text position ϕ_1^ζ to the column under text position ζ in all copies of the pattern are identical. This means that for all $\phi_\alpha^\zeta \in \Phi^\zeta$, $\phi_\alpha^\zeta - \phi_1^\zeta$ are periods of $\mathcal{P}[1..\zeta - \phi_1^\zeta + 1]$. $\Box$

The number of members without credit in the set Φ^ζ is crucial for the analysis of our algorithms since comparisons are charged to the special fund only if a mismatch occurs when the set Φ^ζ has members without credit. Members without credit are introduced to the set Φ only in a backward step to fill holes. We refer to these backward steps as landmarks and denote by $\overline{\phi}$ the first member of Φ which is a text position at which the algorithm was trying to verify an occurrence of the pattern in the last backward step. We also define a *round* to start in a forward step and to end after a backward step is completed or when the end of the string is reached.

Our analysis will bound the number of members without credit that the set Φ can have and show that each time a comparison is charged to the special fund the number of these members decreases. In all other cases during a round this number can not increase. To bound the total number of comparisons charged to the special fund throughout the algorithm we will show that at each round, if any comparison is charged to the special fund, then the set Φ is shifted forward by at least $\frac{m}{2}$ positions. Note that we start with the smallest member of the set Φ equal to 1 and when it becomes larger then $n - m + 1$ the algorithm terminates.

The following lemmas provide bounds on the number of members of Φ^ζ without credits after a backward step.

Lemma 2.3 *Let $\overline{\phi}$ be the smallest member in the set Φ^ζ before a backward step to fill holes and verify an occurrence of the pattern is performed. Let t be the text position of the first mismatch, if a comparison to verify a hole has failed, and let $t = \overline{\phi}$ if all comparisons succeeded. Then all text positions that are smaller or equal to t are removed from the set Φ^ζ and the remaining members of set Φ^ζ are the periods of the pattern that are larger than or equal to $t - \overline{\phi} + 1$, except m, shifted by $\overline{\phi}$ positions.*

Proof: The proof follows immediately from Lemma 2.2 and the modifications to the set Φ^ζ in a backward step to fill holes. □

Corollary 2.4 *Immediately after a backward step is performed, the set Φ is shifted forward by at least π_1 positions. That is, $\phi_1^\zeta - \overline{\phi} \geq \pi_1$. Furthermore, the number of members of Φ^ζ (and therefor also the number of members without credit) is smaller than or equal to $m - \pi_1$.*

Proof: Immediate from Lemma 2.3. □

By Corollary 2.4 if the period of the pattern is at least half of its length, that is $m \leq 2\pi_1$, then the set Φ is shifted forward by at least $\frac{m}{2}$ positions after the backward step. We show next that such a shift is also obtained in the other case after the first subsequent charge to the special fund.

Lemma 2.5 *If $\pi_1 < \zeta - \phi_1^\zeta + 1$ and a comparison between text position ζ and pattern position $\zeta - \phi_1^\zeta + 1$ in a forward step fails, the set Φ^ζ is modified and some members including the smallest member, ϕ_1^ζ, are removed. The new member ϕ_1^ζ satisfies $\zeta - \phi_1^\zeta + 1 \leq$*

π_1. Furthermore, after the construction of Φ^ζ is complete, the number of members of Φ^ζ without credit is smaller than π_1.

Proof: Before the comparison is performed, the members of $\Phi^\zeta - \{\zeta\}$ correspond to periods of the prefix of the pattern $\mathcal{P}[1..\zeta - \phi_1^\zeta]$, except $\zeta - \phi_1^\zeta$, shifted by ϕ_1^ζ positions by Lemma 2.3. Assume that after the comparison has failed, some $\phi_\epsilon^\zeta < \zeta - \pi_1 + 1$ becomes the smallest member of Φ^ζ. By invariant two $\mathcal{P}[\zeta - \phi_1^\zeta + 1 - \pi_1] = \mathcal{P}[\zeta - \phi_\epsilon^\zeta + 1 - \pi_1]$. Since π_1 is a period of any prefix of the pattern also $\mathcal{P}[\zeta - \phi_1^\zeta + 1] = \mathcal{P}[\zeta - \phi_\epsilon^\zeta + 1]$. But $\mathcal{P}[\zeta - \phi_1^\zeta + 1] \neq \mathcal{T}[\phi_1^\zeta]$ and ϕ_ϵ^ζ must also be removed from the set Φ^ζ; a contradiction that shows that all $\phi_\epsilon^\zeta < \zeta - \pi_1 + 1$ are removed from Φ^ζ.

Since $\zeta - \phi_1^\zeta + 1 \leq \pi_1$, the total number of members in Φ^ζ is smaller than or equal to π_1. By a careful examination of the construction of Φ^ζ in Section 2.2.1 we see that after the construction is complete, if $\zeta \in \Phi^\zeta$ it either has a credit or it replaces a member of Φ^ζ that does not have a credit. Therefore, the number of members of Φ^ζ without credit is strictly smaller than π_1. $\Box$

Lemma 2.6 *If $\zeta - \phi_1^\zeta + 1 \leq \pi_1$, then the set Φ^ζ was shifted forward by at least $\frac{m}{2}$ positions since the last backward step. That is, $\phi_1^\zeta - \overline{\phi} \geq \frac{m}{2}$.*

Proof: Before the backward step to fill holes was executed we had $\overline{\phi} + m = \zeta + 1$. Since then, ζ has been increased and also $\zeta - \phi_1^\zeta + 1 \leq \pi_1$. Therefore, the smallest member of Φ^ζ was shifted forward by $\phi_1^\zeta - \overline{\phi} \geq m - \pi_1$ positions. But by Corollary 2.4 we know that the shift is at least π_1 so we must have a shift of at least $\max(\pi_1, m - \pi_1) \geq \frac{m}{2}$. $\Box$

After a backward step, if the set Φ^ζ was not shifted by at least $\frac{m}{2}$ positions, all the new versions of our algorithm resume executing the algorithm described in Section 2.2.1 doing the comparisons in the following way until a mismatch or the next backward step is reached. This has an important advantage as the next lemma shows.

> If a comparison has to be made when there are members of Φ^ζ without credits, the algorithms always compares to the symbol aligned with text position ζ in the first copy of the pattern aligned starting at text position ϕ_1^ζ. This corresponds to the choice $\delta = 1$ in Figure 2.2.

Lemma 2.7 *If all comparisons that are performed between text position ζ and the symbol under it in the first copy of the pattern aligned starting at text position ϕ_1^ζ until we reach the next backward step succeed, then no comparison was charged to the special fund. Otherwise, after the first comparison fails we are guaranteed that the set Φ was shifted forward at least by $\frac{m}{2}$ positions since the last backward step. Furthermore, the number of members in Φ^ζ is at most $\frac{m}{2}$.*

Proof: Comparisons are charged to the special fund only if they fail and some members of Φ^ζ do not have credits. If all comparisons succeed, no comparison is charged to the special fund.

After the first comparison fails, by Lemma 2.5, $\zeta - \phi_1^\zeta + 1 \leq \pi_1$. By Lemma 2.6 the set Φ^ζ was shifted by at least $\frac{m}{2}$ position since the last backward step.

The number of members in Φ^ζ immediately after a backward step is smaller than or equal to $m - \pi_1$ by Corollary 2.4. If Φ^ζ was shifted by more than $\frac{m}{2}$ positions then it has less than $\frac{m}{2}$ members. Otherwise, $\pi_1 \leq \frac{m}{2}$ and after the first mismatch by Lemma 2.6 the number of members of Φ^ζ is smaller than $\pi_1 \leq \frac{m}{2}$. □

At the beginning of each round after a shift of Φ^ζ by $\frac{m}{2}$ text positions is guaranteed (either immediately or after the first mismatch in a forward step) we resume the execution of the algorithm in Figure 2.2 with a choice of δ that depends on the version of the algorithm.

2.2.3 A Binary Pattern

In this section we propose a simple version of our algorithm that gives slightly better bounds when the pattern is over a constant alphabet. We first describe how this version of the algorithm works for patterns over a binary alphabet and then generalize it for any constant alphabet.

Recall, that the number of comparisons charged to text positions is accounted for in Section 2.2.1 and is not larger than n. We need only to bound the number of comparisons charged to the special fund.

Our first goal is to show that the set Φ is shifted forward by at least $\frac{m}{2}$ positions. This can be achieved at the cost of at most one mismatch charged to the special fund by Lemma 2.7. After this goal is achieved the number of members of Φ^ζ (without credits) is guaranteed to be at most $\frac{m}{2}$. After the shift of $\frac{m}{2}$ position is guaranteed we proceed with the algorithm described in Figure 2.2 in a manner that will reduce the number of members of Φ without credit by half each time there is a mismatch.

> We look at the column under text position ζ in copies of the pattern aligned starting at all text position in Φ^ζ that do not have credit and compare text position ζ with a symbol that appears in the largest number of copies.

Lemma 2.8 *A mismatch reduces the number of members without credit in the set Φ^ζ by at least a factor of two.*

Proof: Since the pattern alphabet is binary, if a comparison fails at least half of the members without credit are removed from Φ^ζ. More members without credit can be added to the set Φ^ζ only in a backward step and will be "used" only in the next round. □

Theorem 2.9 *This algorithm performs at most $n + \frac{2\log m+2}{m}(n-m)$ comparisons.*

Proof: Since in each round, if there is a mismatch that is charged to the special fund the set Φ is shifted forward by at least $\frac{m}{2}$ positions, we can have at most $\frac{n-m}{m/2}$ such rounds.

Each round starts with at most one mismatch that is charged to the special fund. After that, there can be no more than $\log m$ comparisons charged to the special fund since each time a comparison is charged to the special fund the number of members of Φ^ζ without credit is decreased by at least a factor of 2. The total number of comparisons charged to the special fund is therefore bounded by $(1+\log m)\frac{2(n-m)}{m}$. If we count also the comparisons charged to text positions, the bound becomes $n + \frac{2\log m + 2}{m}(n-m)$. $\square$

If the size of the pattern alphabet is a larger constant $\sigma(\mathcal{P}) = |\Sigma_\mathcal{P}|$, a similar argument gives us a bound of $n + \frac{2\log_{\sigma(\mathcal{P})/(\sigma(\mathcal{P})-1)} m + 2}{m}(n-m)$.

2.2.4 A General Pattern

In this section we present a version of our algorithm that works for patterns over any alphabet. In each round, at the cost of at most a single mismatch that is charged to the special fund the pattern is shifted forward by at least $\frac{m}{2}$ positions. After the shift is guaranteed we follow two different rules that guide the way that comparisons are performed. The first rule is applied until the number of the remaining members of Φ^ζ without credit is smaller than $\log m$ or until the end of the round. Then, the second rule is applied until there are no members without credit left in Φ^ζ or until the end of the forward step.

Note that a backward step can be reached during the execution of the algorithm while the comparisons are guided either to guarantee a shift of Φ by at least $\frac{m}{2}$ positions or by the two rules and the execution of the rules will restart in a new round after the backward step is completed.

We maintain a text position $\chi \in \Phi^\zeta$ that will guide the choice of the comparison to be performed while following the first rule. We start with $\chi = \phi_1^\zeta$ at the beginning of a new round after a Φ is shifted forward by at least $\frac{m}{2}$ positions. The only members of Φ^ζ that are considered in the first rule for the comparison to be performed are those that are larger than or equal to χ. Members of Φ^ζ that are smaller than χ will be considered only by the second rule.

When χ becomes the largest member of Φ^ζ we proceed to the second rule. Let ϕ_ϵ^ζ be the smallest member of Φ^ζ that is larger than χ. By Lemma 2.2 the members of Φ^ζ correspond to periods of a prefix of the pattern and therefore have some special structure that will be used.

The first rule will repeat the following step until it finds a period that repeats twice in the prefix of the pattern of length $\zeta - \chi + 1$. This step is applied in the beginning of each round after the shift of Φ by $\frac{m}{2}$ positions is guaranteed, starting with $\chi = \phi_1^\zeta$, and after each comparison is performed by the first rule.

> If the period $\overline{\pi} = \phi_\epsilon^\zeta - \chi$ of the prefix of the pattern $\mathcal{P}[1..\zeta - \chi + 1]$ does not repeat twice in this prefix, that is if $\phi_\epsilon^\zeta + \overline{\pi} > \zeta$, we set $\chi = \phi_\epsilon^\zeta$, increment ϵ by one and repeat until either χ is the largest member of Φ^ζ or until the period is repeated twice (i.e. $\phi_\epsilon^\zeta + \overline{\pi} \leq \zeta$).

If χ is the largest member of Φ^ζ then we proceed with the second rule, otherwise we continue with the first rule.

Lemma 2.10 *The members of the set $\Phi^\zeta = \Phi^{\zeta-1} \cup \{\zeta\}$ that are larger then χ correspond to periods of the prefix of the pattern $\mathcal{P}[1..\zeta-\chi]$. Some of these periods might continue to periods of $\mathcal{P}[1..\zeta-\chi+1]$.*

Let ϕ_ϵ^ζ be the smallest member of Φ^ζ that is larger than χ and let $\overline{\pi} = \phi_\epsilon^\zeta - \chi$. The following properties are satisfied at the beginning when $\Phi^\zeta = \Phi^{\zeta-1} \cup \{\zeta\}$ and during any step of the construction of Φ^ζ:

1. *$\phi_\epsilon^\zeta + \delta\overline{\pi} \in \Phi^\zeta$ for non-negative integers values of δ such that $\phi_\epsilon^\zeta + \delta\overline{\pi} < \zeta$.*

   The symbols in the column under text position ζ in the copies of the pattern that are aligned starting at all these text positions are the same.

2. *All other members of Φ^ζ that are larger than χ are also larger than $\zeta - \overline{\pi}$.*

Proof: Since $\mathcal{P}[1..\zeta-\chi]$ has period length $\overline{\pi}$, $\mathcal{P}[i] = \mathcal{P}[i-\overline{\pi}]$ for $\overline{\pi} < i \leq \zeta - \chi$. The proof follows from simple properties of periods.

1. Since the prefix $\mathcal{P}[1..\zeta-\chi]$ has period length $\overline{\pi}$ it has also periods of any integral multiple of $\overline{\pi}$. Some of the periods of $\mathcal{P}[1..\zeta-\chi]$ might not extend to $\mathcal{P}[1..\zeta-\chi+1]$.

   But since $\zeta - \phi_\epsilon^\zeta < \zeta - \chi$, we conclude that $\mathcal{P}[\zeta - \phi_\epsilon^\zeta - \delta\overline{\pi} + 1] = \mathcal{P}[\zeta - \phi_\epsilon^\zeta + 1]$. This means that either all periods that are multiples of $\overline{\pi}$ extend to $\mathcal{P}[1..\zeta-\chi+1]$ or all do not extend.

2. Assume ϕ_ν^ζ is the smallest member of Φ^ζ such that $\chi < \phi_\nu^\zeta < \zeta$ and is not of the form $\phi_\epsilon^\zeta + \delta\overline{\pi}$. Since $\chi + \overline{\pi} = \phi_\epsilon^\zeta < \phi_\nu^\zeta$, if $\phi_\nu^\zeta \leq \zeta - \overline{\pi}$ then $\mathcal{P}[\phi_\nu^\zeta - \chi - \overline{\pi} + 1..\phi_\nu^\zeta - \chi] = \mathcal{P}[\phi_\nu^\zeta - \chi + 1..\phi_\nu^\zeta - \chi + \overline{\pi}]$ and also $\phi_\nu^\zeta - \overline{\pi} \in \Phi^\zeta$; contradiction to the minimality of ϕ_ν^ζ.

□

The first rule performs comparisons in the following manner:

> At this point it is guaranteed that the period $\overline{\pi} = \phi_\epsilon^\zeta - \chi$ of the prefix of the pattern $\mathcal{P}[1..\zeta-\chi]$ repeats twice in this prefix.
>
> The comparison is performed between text position ζ and the symbol aligned with this text position in the copy of the pattern that is aligned starting at ϕ_ϵ^ζ. This corresponds to the choice $\delta = \epsilon$ in Figure 2.2.
>
> The set Φ^ζ is modified if necessary according to the outcome of the comparison and the loop above is repeated to update χ.

- If the comparison succeeded χ will be updated only if its current value was removed from the set Φ^ζ. Because the comparison succeeded χ will be assigned to the previous value of ϕ_ϵ^ζ that is still in Φ^ζ. In this case the loop above has to be repeated to update χ.

  Note that if χ was not removed from Φ^ζ then the period of $\mathcal{P}[1..\zeta-\chi+1]$, $\overline{\pi}$, still repeats twice. If χ was removed, the period may or may not repeat twice.

- If the comparison failed, χ may or may not have been removed from Φ^ζ but will be updated in any case. The new value of χ is set to the smallest surviving member of Φ^ζ that is larger than χ. If such a member does not exist we proceed to the second rule. Note that by Lemma 2.10 all members of the form $\phi_\epsilon^\zeta + \delta\overline{\pi}$ are removed from Φ^ζ and the new value of χ is larger than $\zeta - \overline{\pi}$.

Lemma 2.11 *The number of members of Φ^ζ that are smaller than χ is less than* $\log m$ *at any point while the first rule is guiding the comparisons to be performed. The number of mismatches that can occur is also smaller than* $\log m$.

Proof: After each mismatch χ is updated. Since the number of members of Φ^ζ that are smaller than χ increases only when χ is updated, we have to consider only these occasions. Note that the number of members of Φ^ζ that are smaller than χ can be reduced at any time. We prove that even if none is removed the bound still holds.

We show that each time χ is updated the potential length of a period of $\mathcal{P}[1..\zeta-\chi+1]$ that repeats twice is halved. At the beginning the potential length for such a period is $\frac{m}{2}$.

Immediately after a mismatch has occurred, $\zeta - \chi < \overline{\pi}$ and the longest period that repeats twice that can be found must be shorter than $\frac{\overline{\pi}}{2}$. This means that at most one member of Φ could be added to the set of members that are smaller than χ at the cost of halving the length of a potential period that repeats twice.

Now, when the loop is executed to find such a period that repeats twice, each time the value of χ is modified, one member of Φ becomes smaller than χ. But since the period did not repeat twice, $\zeta - \chi$ is halved and the bound on a potential period that can repeat twice is also halved.

Therefore, the total number of mismatches while working under the first rule and the number of members of Φ^ζ that are smaller than χ are bounded by $\log m$. $\Box$

The second rule is applied when χ is the largest member of Φ^ζ. At the beginning when we start applying the second rule, the number of members of Φ^ζ without credit is smaller than $\log m$ by Lemma 2.11.

> The comparison is performed between text position ζ and the symbol aligned with this text position in the copy of the pattern that is aligned starting at ϕ_1^ζ. This corresponds to the choice $\delta = 1$ in Figure 2.2.

Theorem 2.12 *The algorithm performs at most* $n + \frac{4\log m+2}{m}(n-m)$ *comparisons.*

Proof: As in Theorem 2.9 there are at most $\frac{n-m}{m/2}$ rounds that have comparisons charged to the special fund. After the first mismatch is charged to the special fund to guarantee a shift of Φ by at least $\frac{m}{2}$ positions, by Lemma 2.11 there can be at most $\log m$ comparisons charged to the special fund while following the first rule and at most $\log m$ comparisons charged to the special fund while following the second rule. Therefore, the number of comparisons charged to the special fund is bounded by $\frac{4\log m+2}{m}(n-m)$ and the total number of comparisons performed by the algorithm is bounded by $n + \frac{4\log m+2}{m}(n-m)$.
□

2.2.5 Colussi's Algorithm

In this section we describe Colussi's [32] algorithm and discuss the relation between that algorithm and a similar version of our algorithm.

Colussi [32] calls a pattern position h a *hole* if all periods of $\mathcal{P}[1..h-1]$ continue to $\mathcal{P}[1..h]$. That is, a pattern position h is a hole if $\Pi^{\mathcal{P}[1..h-1]} \subset \Pi^{\mathcal{P}[1..h]}$ or alternatively, a pattern position h is a hole if there is no prefix period of the pattern that is terminated at position h. Note that holes in Colussi's algorithm are associated with pattern positions, while holes in our algorithm are associate with text positions. For each pattern position α that is not a hole, Colussi defines $kmin(\alpha)$ to be the smallest terminated period at position α, that is $kmin(\alpha) = \min\{\pi | \pi \in \Pi^{\mathcal{P}[1..\alpha-1]} \ and \ \pi \notin \Pi^{\mathcal{P}[1..\alpha]}\}$.

Based on this definition of holes Colussi defines a sequence h_i, $i = 1..m$, that will dictate the order in which pattern symbols are compared to text symbols. The sequence h_i is defined to include first all pattern positions that are not holes in an increasing order following by the pattern positions that are holes in a decreasing order.

Colussi's algorithm proceeds by aligning the pattern with a text position and comparing the pattern positions in the order they are listed in the h_i sequence to the text positions that are aligned with them. If a comparison fails the pattern is shifted forward to be aligned at the nearest position of the text that can still start an occurrence of the pattern. The next text position that the pattern will be aligned with in case of a mismatch is determined by the following rules:

- If the comparison that failed was at a pattern position h_ϵ that is not a hole, the pattern is shifted forward by $kmin(h_\epsilon)$ positions. This means that the pattern will be aligned at a position of the text that its shortest period that is terminated at h_ϵ was aligned with before. Note, that such a terminated period must exist by the definition of a hole.

  Since in Colussi's algorithm holes are associated with pattern templates and not with text positions the holes are also shifted. Colussi shows that there is no need to worry about shifting the holes since a text position that was aligned with a pattern hole before will still be aligned with a hole. However, some comparisons are "forgotten" since text positions that were known before can be covered by a hole after the shift.

- If the comparison that failed was at a pattern position h_ϵ that is a hole, the pattern is shifted by its shortest period that is larger than h_ϵ. Note, that by the definition of the h_i sequence, all text positions that were aligned with pattern positions larger than h_ϵ have been compared to successfully before, and need not be compared again. This allows Colussi's algorithm to shift the pattern without having to go back to compare symbols.

```
– The sequence h_i lists first the ν pattern positions that are not holes followed
–     by the m − ν positions that are holes. Define also h_{m+1} = 0.
– β is the text position that the pattern is aligned with.
– ε is the index to the h_i sequence.
– θ is a text position behind which there is no need to compare.
    β = 1
    ε = 1
    θ = 0
    while β ≤ n − m + 1 do
        while ε ≤ m and θ < β + h_ε − 1 and P[h_ε] = T[β + h_ε − 1] do
            ε = ε + 1
        if ε = m + 1 or θ ≥ β + h_ε − 1 then
            An occurrence of the pattern starting at text position β is found.
        if ε > ν then    – A mismatch in a hole, or no mismatch at all.
            θ = β + m − 1
            β = β + min{π|π ∈ Π^P  and  π ≥ h_ε}
            ε = min{i|h_i > θ − β + 1}
        else
            β = β + kmin(h_ε).
            ε = min{i|h_i ≥ h_ε − kmin(h_ε)}
        end
    end
```

Figure 2.3: Colussi's algorithm.

In both cases the algorithm resumes comparing symbols in the order dictated by the h_i sequence starting from the first h_i that is aligned with a text position that larger than the largest text position compared. Colussi's algorithm is described in Figure 2.3.

2.2.6 A Similar Version of Our Algorithm

In the description of the new algorithm in Section 2.2.1, we left the choice of the symbol that a text position ζ is compared with, if any member of the set Φ^ζ, except ζ, does not have a credit, to the particular version of the algorithm. In the version described in this section we choose always to compare text position ζ to the symbol aligned with it in the first copy of the pattern that is aligned starting at text position ϕ_1^ζ. This corresponds to the choice of $\delta = 1$ in Figure 2.2.

There are two major differences between this version of our algorithm and Colussi's [32] algorithm. These differences are related to what happen when a comparison to a pattern position that is not a hole fails. Note, that in this case, the shift performed by Colussi's algorithm corresponds to the removal of ϕ_1^ζ and possibly other members from Φ^ζ. Comparisons to pattern positions that are holes are handled identically to the backward step to fill holes in our algorithm.

- When a comparison fails, Colussi's algorithm shifts the pattern few positions ahead. Since holes in Colussi's algorithm correspond to pattern positions the holes are also shifted. Some holes can be aligned with text positions that have been compared to successfully earlier, and these comparisons will be "forgotten".

  This obliviousness of Colussi's algorithm can be easily fixed by associating information about holes also with text positions or by using a buffer.

- When a comparison to pattern position h_α fails, Colussi's algorithm shifts the pattern by $kmin(h_\alpha)$ positions. This corresponds to deleting from Φ^ζ all ϕ_β^ζ such that $\phi_\beta^\zeta < \phi_1^\zeta + kmin(h_\alpha)$.

  In our algorithm, since we have complete information about the pattern and we do not care about efficient implementation on a conventional model, we remove from Φ^ζ at least all text positions that are removed by Colussi's algorithm and possibly more. Colussi's algorithm might repeat some comparisons when it shifts over these text positions.

  This phenomena is demonstrated by the pattern '*aabaaa*' when a comparison is performed at the last pattern symbol. Note that in this case, the set $\Phi^\zeta = \{\phi_1^\zeta, \phi_1^\zeta + 3, \phi_1^\zeta + 4, \phi_1^\zeta + 5\}$. If the comparison to the last pattern position fails, our algorithm knows that the symbol under that column in the copies of the pattern aligned starting at $\phi_1^\zeta + 4$ and $\phi_1^\zeta + 5$ is also an 'a' and these text positions must be removed from the set Φ^ζ. Colussi's algorithm on the other hand might still consider these text position as a possible start of an occurrence.

  This obliviousness of Colussi's algorithm can not be easily fixed by using a buffer.

There is a third difference between Colussi's algorithm and our algorithm that is not realized in this version of our algorithm:

- Our algorithm does not necessarily have to perform comparisons which would eliminate ϕ_1^ζ in case of a mismatch as the version described in this section does.

We are now ready to analyze this version of our algorithm.

Recall, that the number of comparisons that are charged to text positions is accounted for in Section 2.2.1 and is not larger than n. We only need to bound the number of comparisons charged to the special fund. Comparisons are charged to the special fund only in case of a mismatch when the set Φ has members without credit. Since members without credits are added to the set Φ only in a backward step to fill holes, we refer to each such backward step as a landmark, and analyze separately the number of comparisons charged to the special fund between two backward steps.

Theorem 2.13 *The number of comparisons that are charged to the special fund is smaller than* $\frac{\min(\pi_1^{\mathcal{P}}, m-\pi_1^{\mathcal{P}})}{m}(n-m)$ *and thus, total number of comparisons performed is bounded by* $n + \frac{\min(\pi_1^{\mathcal{P}}, m-\pi_1^{\mathcal{P}})}{m}(n-m)$.

Proof: Assume that after a backward step the set Φ^ζ was shifted forward by δ positions. Thus the size of Φ^ζ and the number of members it has without credit is at most $m-\delta$. Since each charge to the special fund reduces this number and causes a shift of Φ^ζ (ϕ_1^ζ is deleted), the number of comparisons charged to the special fund till the next backward step (or the end of the algorithm if there is no subsequent backward step) is $\epsilon \leq m-\delta$ and the total shift is at least ϵ. Adding both shifts we get a shift of at least $\delta+\epsilon$ or at most $\frac{\epsilon}{\delta+\epsilon}$ charge to the special fund per unit of shift. This function is maximized for the maximal value of ϵ (because of monotonicity), and thus bounded above by $\frac{m-\delta}{m}$. By Lemma 2.3 and Lemma 2.5 $\delta \geq \max(\pi_1, m-\pi_1)$ and the charge per unit of shift is bounded above by $\frac{\min(\pi_1, m-\pi_1)}{m}$.

The proof now follows from the fact that the total shift of Φ throughout the algorithm is $n-m$ and by adding the n comparisons charged to the text symbols. $\Box$

2.2.7 Galil and Giancarlo's Algorithm

Galil and Giancarlo [48] noticed that Colussi's [32] algorithm performs very badly if the pattern starts and ends with a sequence of repetitions of the same symbol. For these patterns Colussi's algorithm shifts by a single position and $\frac{3}{2}n$ comparisons are actually performed. Galil and Giancarlo devised a way to avoid these shifts by a single position. Their algorithm is very similar to Colussi's algorithm and to the algorithm described in the previous section. The improvement to Colussi's algorithm is achieved by performing the comparisons in a different way at certain steps of the algorithm that correspond to the the choice of $\delta = 2$ in Figure 2.2. The same change applied to our algorithm will give their bounds.

2.3 Open Problems

The exact complexity of string matching is not determined yet and there are several open problems left. Although the presentation in this section is purely theoretical, there is a good chance that by studying the exact complexity of the string matching problem we can gain better understanding of the structure of the problem and obtain fast practical algorithms.

1. What is the exact number of comparisons that are required by an on-line algorithm? By an off-line algorithm? By a l-look-ahead algorithm? The upper bounds given in this chapter most probably can be improved.

2. Is a finite look-ahead sufficient? Namely, is there a finite value of l for which $c^l_{\mathcal{P}}(n) = c^{off-line}_{\mathcal{P}}(n)$?

3. Galil and Giancarlo [48, 49] consider also the number of comparisons required by algorithms that only test if there is an occurrence of the pattern in the text and do not find all occurrences. What are the bounds for these type of problems?

4. Do algorithms that can compare pairs of text symbols have an advantage over algorithm that can compare only a pattern symbol to a text symbol?

5. What is the exact number of comparisons required to find the period of a string? To find all periods?

6. What are the exact bounds of the string matching problem if the pattern preprocessing is accounted for?

Chapter 3

Parallel String Matching Algorithms

In parallel computation, one has to be more careful about the definition of the problem. For the string matching problem we assume that the input strings $\mathcal{P}[1..m]$ and $\mathcal{T}[1..n]$ are stored in memory and the required output is a Boolean array $\mathcal{M}[1..n]$ which will have a 'true' value at each text position where the pattern occurs and 'false' value where there is no occurrence. The importance of this definition is demonstrated by the following theorem:

Theorem 3.1 *It is impossible to* count *the number of occurrence of the pattern in the text in less than $\frac{\log n}{\log\log n}$ time on a CRCW-PRAM with any polynomial number of processors.*

Proof: We show a reduction to the lower bound given in Theorem 1.13 for computing the parity of n input bits.

Assume the pattern string is '1' and that the n input bits are presented as the text string. If the number of occurrences is computed, then the number of input bits that are '1' is known. The parity is the number of these bits modulo two. Thus, if the number of occurrences is known the parity is computed immediately. □

The last theorem shows that it is impossible to have a fast CRCW-PRAM algorithm that stores the occurrences of the pattern in a compressed array, where all the occurrences are listed immediately and explicitly at the beginning of the array. Such a representation is useful sometimes and can be computed from our representation in $O(\frac{\log n}{\log\log n})$ time on a $\frac{n\log\log n}{\log n}$ processor CRCW-PRAM [30]. Another possible representation that can be computed even more efficiently by algorithms of Berkman and Vishkin [18] and Ragde [78] is a linked list of all the occurrences.

Note that Theorem 3.1 does not imply that the string matching problem as we defined it cannot be solved faster. In fact, a string matching algorithm that is described later solves the problem in optimal[1] $O(\log\log n)$ time and a very simple algorithm that solves

[1] Here, optimal means that the total number of operations used is linear, since there exist sequential linear time algorithms for the string matching problem.

the string matching problem in constant time using an nm processor CRCW-PRAM is described next. The situation is similar for all the string problems that we develop efficient parallel algorithm for in this dissertation. By choosing the right representation of the output, it is possible to get around the lower bound of Theorem 1.12 and to solve all these problems by trivial parallel algorithms that use a polynomial number of processors. Optimal solutions are usually more difficult to obtain and make up a major part of this dissertation. There are many other problems that have fast parallel algorithms that get around the lower bound of Theorem 1.13 [16, 17, 20, 67, 83, 86].

Lemma 3.2 *The string matching problem can be solved in constant time by an nm processor CRCW-PRAM.*

Proof: Consider each of the first $n-m+1$ text positions as possible starts of an occurrence and initialized the output Boolean array $\mathcal{M}$ to indicate that these text positions are occurrences. Note that these are the only text positions that can start occurrences of the pattern.

Assign m processors to each such position to verify the occurrence that might start there. Verifying an occurrence is simple; perform m comparisons in parallel and any mismatch will change the appropriate value of the $\mathcal{M}$ array to indicate that there is not occurrence of the pattern at that text position. □

One can get better lower bounds for a CREW-PRAM. These lower bounds hold also for our representation of the problem as the next theorem shows:

Theorem 3.3 *It is impossible to solve the string matching problem in less than* $\log n$ *time on a CREW-PRAM with any number of processors.*

Proof: We show a reduction to the lower bound given in Theorem 1.12 for computing the Boolean AND function of n input bits.

Assume the pattern string is '1^n' and that the n input bits are presented as the text string. Note that the pattern and the text are of the same length and there can be only one occurrence. If there is an occurrence of the pattern in the text, then all the input bits are 1 and the Boolean AND function is 1. In any other case the Boolean AND function of the inputs bits is 0. Thus, if the string matching problem was solved the Boolean AND function was computed. □

We show next that the time bounds for CRCW-PRAM string matching algorithms can always be expressed as functions of m, the length of the pattern, rather then as functions of n, the length of the text. This means that without loss of generality we can always assume that the text string is only a constant times longer than the pattern string and that our bounds can be written in terms of n or m.

Lemma 3.4 *If the string matching problem can be solved in $T(n)$ time using a total of n operations it can also be solved in $O(T(m))$ time with the same number of operations.*

Proof: We can partition the text string into n/m overlapping blocks of length $2m$ each in such way that by solving the string matching problem for each block separately we can get all the occurrences of the pattern in the text. It takes $T(2m)$ time to solve the string matching problem in each block, but all blocks can be processed simultaneously what gives us the required bound. $\Box$

In this chapter we describe optimal parallel algorithms and lower bounds for string matching and for finding all periods of a string on the CRCW-PRAM model. We obtain a bound of $\Theta(\log\log n)$ time on $\frac{n}{\log\log n}$ processors and a bound of $\Theta(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time on p processors.

3.1 Historical Overview

The first optimal parallel string matching algorithm was discovered by Galil [46]. It solves the problem over a constant alphabet in $O(\log m)$ time on a $\frac{n}{\log m}$-processor CRCW-PRAM. This algorithm uses the four-Russions method and packs many input symbols into one register to compare them in a single operation. Galil's [46] algorithm was soon improved by Vishkin [87] to an algorithm that works over a general alphabet.

Breslauer and Galil [25] obtained an optimal $O(\log\log m)$ time algorithm for a general alphabet. This algorithm is based on the algorithms of Galil [46] and Vishkin [87]. The algorithm that was a part of my Master's Thesis [24] is described later for completeness. An analysis of the algorithm for any number of available processors that was not included in the original work is also presented.

The simplest known parallel string matching algorithm is perhaps the optimal $O(\log n)$ time randomized algorithm of Karp and Rabin [60]. This algorithm works under the constant alphabet assumption and is also the only optimal EREW-PRAM parallel algorithm known. All other optimal algorithms require the more powerful CRCW-PRAM model.

Other parallel algorithms based on Karp, Miller and Rosenberg's [59] method [40, 62] also work in $O(\log n)$ time over a fixed alphabet. This method is useful for many other string problems but it seems to require $O(\log n)$ time and a larger memory space.

Vishkin [88] developed an optimal $O(\log^* m)^2$ time algorithm. Unlike the case of the other algorithms this time bound does not account for the preprocessing of the pattern. The preprocessing in Vishkin's algorithm takes optimal $O(\frac{\log^2 m}{\log\log m})$ time. Vishkin's super fast algorithm raised the question whether an optimal constant-time algorithm is possible. We partially settle this question by giving a lower bound of $\Omega(\log\log m)$ for parallel string matching over a general alphabet. The lower bound proves that a slower preprocessing is crucial for Vishkin's algorithm.

Vishkin's algorithm was improved recently by Galil [47] who discovered a constant time optimal algorithm. Galil's [47] new algorithm requires the same time as Vishkin's algorithm for the preprocessing step.

---

[2]The function $\log^* m$ is defined as the smallest k such that $\log^{(k)} m \leq 2$, where $\log^{(1)} m = logm$ and $\log^{(i+1)} m = \log\log^{(i)} m$.

Apostolico [6] also designed a constant time optimal parallel string matching algorithm over an ordered alphabet. Apostolico's algorithm uses the alphabet order to compute a special representation for the input strings in a preprocessing step that takes $O(\log n)$ time and is not optimal. There is no distinction between the pattern and the text in the preprocessing step of Apostolico's algorithm and it has the advantage that it can report occurrences of any substring of the preprocessed strings in any other substring.

3.2 An $O(\log\log n)$ Time Algorithm

In this section we present an $O(\log\log n)$ time $\frac{n}{\log\log n}$-processor CRCW-PRAM string matching algorithm [16, 24, 25]. If the number of available processors is p we show that the algorithm takes $O(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time.

Our algorithm uses the weakest version of the common CRCW-PRAM model: the only write conflict allowed is that processors can write simultaneously the value 1 into a memory cell. The algorithm consists of two phases: the pattern is preprocessed in the first phase to compute some information that is used in the second phase to process the text.

By Lemma 3.2 the string matching problem can be solved in constant time on an nm-processor CRCW-PRAM. Assuming we can eliminate all but l of the possible occurrences (ignoring the problem of assigning the processors to their tasks), we can use the same method to get a constant time parallel algorithm that uses only lm processors. Both Galil [46] and Vishkin [87] use this approach. The only problem is that one can have many occurrences of the pattern in the text, even much more than the n/m needed for optimality in the discussion above.

To eliminate this problem, we use properties of periods as suggested by Galil [46] and also used by Vishkin [87].

Lemma 3.5 *If the pattern $\mathcal{P}[1..m]$ has period length $\pi_1^{\mathcal{P}}$, then there cannot be two occurrences of the pattern at text positions i and j, for $|j-i| < \pi_1^{\mathcal{P}}$.*

Proof: Assume that there is an occurrence of the pattern at text positions i and j. Since $0 < |j-i| < \pi_1^{\mathcal{P}} \leq m$ the pattern must have a period of length $|j-i| < \pi_1^{\mathcal{P}}$. But since $\pi_1^{\mathcal{P}}$ is the shortest period of the pattern, $\pi_1^{\mathcal{P}} \leq |j-i|$; a contradiction. □

By the last lemma, there cannot be more than $n/\pi_1^{\mathcal{P}}$ occurrences of the pattern in the text. If instead of matching the whole pattern, we look only for occurrences of the prefix $\mathcal{P}[1..\pi_1^{\mathcal{P}}]$, assuming we could eliminate many possible occurrences and have only $n/\pi_1^{\mathcal{P}}$ left, we could use the constant time algorithm described above to verify these occurrences using only n processors. Then, by counting the number of consecutive occurrences of $\mathcal{P}[1..\pi_1^{\mathcal{P}}]$, we can find the occurrences of the whole pattern. Later, by Theorem 1.11 we can slow down this constant time string matching algorithm to run in $O(s)$ time on lm/s processors. Theorem 1.11 as well as other computations described below require the assignment of processors to their tasks which in our case is done using standard techniques as in Shiloach and Vishkin's [83] maximum finding algorithm.

3.2.1 Witnesses

An important idea in our algorithm is a method suggested by Vishkin [87], that enables us to eliminate many possible occurrences in constant time. One computes some information about the pattern which is called *the witness array* and denoted by $\mathcal{W}[1..m]$, and uses it in the second phase for the analysis of the text.

Let $0 \le v \le m$ be an integer that is not a period of the pattern. By the definition of a period there must exist an index k such that

$$\mathcal{P}[k] \neq \mathcal{P}[k-v].$$

We call this k a *witness* to the fact that v is not a period and define

$$\mathcal{W}[v+1] = k.$$

We are interested only in $\mathcal{W}[i]$, for $1 < i \le \pi_1^{\mathcal{P}}$. The following lemma shows that we can consider only a short prefix of the pattern if the period of the pattern is repeated many times.

Lemma 3.6 *The witness array $\mathcal{W}[i]$, for $1 < i \le \pi_1^{\mathcal{P}}$, can be based only on the first $2\pi_1^{\mathcal{P}} - 1$ symbols of the pattern.*

Proof: Assume $\mathcal{W}[i]$ is given and some $\mathcal{W}[i]$ is greater than or equal to $2\pi_1^{\mathcal{P}}$. Then by the periodicity properties it can be modified to be in the desired range. Let $\omega = \mathcal{W}[i] \ mod\, \pi_1^{\mathcal{P}}$, then if $\omega < i$ set $\mathcal{W}[i] = \omega + \pi_1^{\mathcal{P}}$, otherwise set $\mathcal{W}[i] = \omega$. The new $\mathcal{W}[i]$ must also be a valid witness and it satisfies the required conditions. □

3.2.2 Duels and Counting

We call the pattern *periodic* if its length is at least twice its period length (i.e. $m \ge 2\pi_1^{\mathcal{P}}$). Having computed the $\mathcal{W}$ array in the pattern preprocessing phase, Vishkin [87] suggests the following method that he calls a *duel* to eliminate close possible occurrences. Suppose we suspect that an occurrence of the pattern may start at text positions i and j, where $0 < j - i < \pi_1^{\mathcal{P}}$. Since we computed $r = \mathcal{W}[j-i+1]$, we can find in constant time a symbol in the text that will eliminate one or both of the possible occurrences. More specifically, since $\mathcal{P}[r] \neq \mathcal{P}[r-j+i]$, at most one of $\mathcal{P}[r]$ and $\mathcal{P}[r-j+i]$ can be equal to $\mathcal{T}[r+i-1]$. See Figure 3.1.

In practice, we eliminate possible occurrences of some prefix of the pattern. In the periodic case, we saw in the previous section that the witness information can be based only on the first $2\pi_1^{\mathcal{P}}$ symbols of the pattern, thus we eliminate positions in which there is no occurrence of $\mathcal{P}[1..2\pi_1^{\mathcal{P}}]$. While in the non-periodic case, the witness information

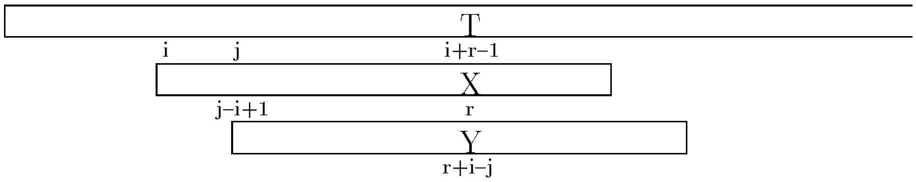

Figure 3.1: $X \neq Y$ and therefore we cannot have $T = X$ and $T = Y$.

is based on the whole pattern and positions where there is no occurrence of it can be eliminated. Having many such duels in pairs, Vishkin's [87] algorithm eliminates enough possible occurrences of $\mathcal{P}[1..\pi_1^{\mathcal{P}}]$ in the text in $O(\log m)$ time and verifies them using the constant time algorithm described above. We manage to reduce the time of Vishkin's [87] algorithm to an $O(\log\log m)$ time algorithm using the following observations:

- Duels "work like" maximum. Having a block of the text of length equal to $\pi_1^{\mathcal{P}}$, only one occurrence of the pattern might start in it. Assume that the pattern can start anywhere within that block, and suppose we have $\binom{\pi_1^{\mathcal{P}}}{2}$ processors. Assign a processor to each pair and perform a duel. Since in every pair at least one loses, at the end we are left with no more than one possible occurrence in each block. The exact details of the algorithm appear in the next sections.

- We simplify the "counting" of consecutive occurrences of $\mathcal{P}[1..\pi_1^{\mathcal{P}}]$ in the text in the periodic case. Theorem 1.13 shows that computing the parity of n bits on a CRCW-PRAM takes $\frac{\log n}{\log\log n}$ with any polynomial number of processors, so no "real" counting is possible within our time bounds. Assume without loss of generality that the text is of length $n = 2m - \pi_1^{\mathcal{P}}$ (divide the text into $\frac{n}{m-\pi_1^{\mathcal{P}}} = O(\frac{n}{m})$ overlapping groups of length $2m - \pi_1^{\mathcal{P}}$ as in Lemma 3.4). We call an occurrence of $\mathcal{P}[1..2\pi_1^{\mathcal{P}}]$ at position i an *initial occurrence* if there is no occurrence of it at position $i - \pi_1^{\mathcal{P}}$. We call such occurrence a *final occurrence* if there is no occurrence at position $i + \pi_1^{\mathcal{P}}$. The main observation is that there is at most one initial occurrence of interest which is the rightmost initial occurrence in the first $m - \pi_1^{\mathcal{P}}$ text positions. Any initial occurrence in a text position greater than $m - \pi_1^{\mathcal{P}}$ is of no interest since there are not enough symbols in the text to match the whole pattern. Since the pattern is periodic with period length $\pi_1^{\mathcal{P}}$ initial occurrences which are smaller cannot start occurrences of the pattern either. The corresponding final occurrence is the smallest final occurrence which is greater than the initial occurrence.

3.2.3 Processing the Text

As we mentioned above, duels are like maxima. We describe an optimal $O(\log\log m)$ time text analysis algorithm that works similarly to the maximum finding algorithm of Shiloach and Vishkin [83] and is based on having $\mathcal{W}[2 \cdots r]$, for $r = \min(\pi_1^{\mathcal{P}}, \lceil m/2 \rceil)$ computed in the pattern analysis phase. In the periodic case we divide the text into groups of length $n = 2m - \pi_1^{\mathcal{P}}$, while in the non-periodic case we work on the whole text.

Without loss of generality we have $\mathcal{W}[i] < 2\pi_1^{\mathcal{P}}$. Partition the text into blocks of length r. We have n/r such blocks. In each block mark all positions as possible occurrences. Partition them into groups of size $\sqrt{r}$ and repeat recursively. The recursion bottoms out with one processor per block of size 1, where nothing is done. When done, we are left with one possible occurrence (or none) in each block of size $\sqrt{r}$, thus $\sqrt{r}$ possible occurrences altogether. Then in constant time make all duels as described above. We are left with a single possible occurrence (or none) in each block of size r.

The algorithm described above takes $O(\log\log m)$ time but is not optimal; it requires n processors. To achieve optimality we first partition our block into small blocks of size $\log\log r$. To each one of the $r/\log\log r$ small blocks assign a processor and make duels between pairs using a sequential algorithm till left with at most one possible occurrence in each small block. Then, proceed with the $O(\log\log r)$ algorithm having at most $r/\log\log r$ possible occurrences to start with. Since we have n/r blocks and in each block we used $r/\log\log r$ processors, we need a total of $n/\log\log r$ processors for this computation. Left with at most n/r possible occurrences, we can use the constant algorithm we described above to verify these occurrences. The next step depends on the periodicity of the pattern, we have two cases:

1. The pattern is not periodic ($m < 2\pi_1^{\mathcal{P}}$, $r = m/2$): Verify the whole pattern at each possible occurrences. This can be done using $\frac{mn}{r} = 2n$ processors in constant time.

2. The pattern is periodic:

   - Verify only the first $2\pi_1^{\mathcal{P}}$ symbols of the pattern at each possible occurrence in the text. This can be done using only $2n$ processors in constant time.
   - Find the initial occurrence and the corresponding final occurrence: First find all initial occurrences and final occurrences. Then, find the maximal initial occurrence in the first $m - \pi_1^{\mathcal{P}}$ symbols and the corresponding final occurrence. This can be done in constant time using m processors on our weak CRCW-PRAM by Theorem 1.14.
   - Let v be a suffix of the pattern $\mathcal{P}[1..m]$ of length $m \ \ mod\, \pi_1^{\mathcal{P}}$. Verify v right after the final occurrence. Note that v occurs after each non-final occurrence since v is a prefix of u.
   - For each verified occurrence of $\mathcal{P}[1..2\pi_1^{\mathcal{P}}]$ check if enough occurrences follow and if followed by a verified occurrence of v. This can be done using the position of the initial occurrence and the final occurrence.

We will use the following technical lemma:

Lemma 3.7 *If $c > 1$ and $p^{1-c^{-k}} = \frac{m}{2}$, then $k = \Theta\left(\log\log_{\frac{2p}{m}} p\right)$.*

Proof: We have $\frac{2p}{m} = p^{c^{-k}}$. Taking logarithms twice we get

$$k = \log_c \frac{\log p}{\log \frac{2p}{m}} = \log_c \log_{\frac{2p}{m}} p.$$

□

Now, we can prove the bound for the text processing phase.

Lemma 3.8 *The text processing phase can be implemented in $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors and in $O(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time on p processors.*

Proof: As we have seen the duels in the text processing phase can be implemented in $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors. Steps 1 and 2 that are described above can be done in constant time using n processors or in $O(\log\log m)$ time using $n/\log\log m$ processors by Theorem 1.11.

It remains to show that if the number of available processors is p, then the algorithm takes $O(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time. If $\frac{p<n}{\log\log n}$, then by Theorem 1.11 the algorithm described above can be slowed down to work on p processors in $O(\frac{n}{p})$ time. If $\frac{n}{\log\log n} \leq p \leq n$ then the bound above is still $O(\log\log n)$. If the number of processors if larger than n^2, then the simple algorithm of Lemma 3.2 solves the problem in constant time. The only remaining case is when $n \leq p \leq n^2$.

In this case we can use larger blocks at each step of the recursion described above. At stage number i of the recursion, the size of the blocks is $\frac{r}{(\frac{pr}{n})^{1-2^{-i}}}$. The recursion terminates after i steps when $(\frac{pr}{n})^{1-2^{-i}} \geq \frac{r}{2}$. By lemma 3.7, $\log\log_{\frac{2p}{n}} \frac{pr}{n}$ steps are necessary. But $n = 2m$ and r can be as large as m so the resulting bound is $O(\log\log_{\frac{2p}{n}} p)$.

When we combine all these bounds into a single expression we get the promised bound of $O(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time. □

3.2.4 Processing the Pattern

The $\mathcal{W}$ array which we used in the text processing stage is computed incrementally. Knowing that some witnesses are already computed in previous stages, one can compute more witnesses easily. Let i and j be two indices in the pattern such that $i < j < \lceil m/2\rceil + 1$. If $s = \mathcal{W}[j-i+1]$ is already computed then we can find at least one of $\mathcal{W}[i]$ or $\mathcal{W}[j]$ using a duel on the pattern as follows:

- If $s+i-1 \leq m$ then $s+i-1$ is also a witness either for i or for j.
- If $s+i-1 > m$ then either s is a witness for j or $s-j+i$ is a witness for i. See Figure 3.2. Note that if i or j are larger than $\frac{m}{2}$ this argument does not hold.

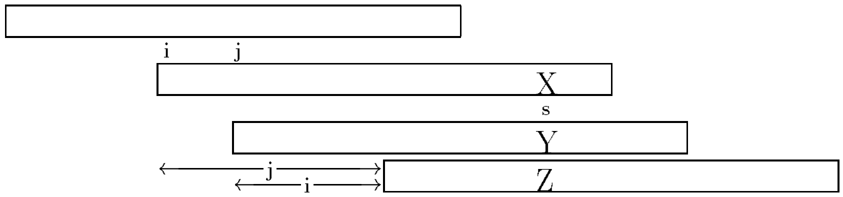

Figure 3.2: $X \neq Y$ and therefore we cannot have $Z = X$ and $Z = Y$.

We first describe an $O(\log\log m)$ non-optimal algorithm. It works in stages and it has at most $\log\log m$ stages. Let $k_i = m^{1-2^{-i}}$, $k_0 = 1$. At the end of stage i, we have at most one yet-to-be-computed witness in each block of size k_i. The only yet-to-be-computed index in the first block is 1.

1. At the beginning of stage i we have at most k_i/k_{i-1} yet-to-be-computed witnesses in the first k_i–block. Try to compute them using the naive algorithm on $\mathcal{P}[1..2k_i]$ only. This takes constant time using $2k_i\frac{k_i}{k_{i-1}} = 2m$ processors.

2. If we succeed in producing witnesses for all the indices in the first block (all but the first for which there is no witness), compute witnesses in each following block of the same size using the optimal duel algorithm described in the text processing section. This takes $O(\log\log m)$ time only for the first stage. In the following stages, we will have at most $\sqrt{m}$ indices for which we have no witness, and duels can be done in constant time.

3. If we fail to produce a witness for some $2 \leq j \leq k_i$, it follows that $\mathcal{P}[1..2k_i]$ is periodic with period length π, where $\pi = j - 1$ and j is the smallest index of a yet-to-be-computed witness. By the periodicity properties mentioned above, all yet-to-be-computed indices within the first block are of the form $k\pi + 1$. Check periodicity with period length π to the end of the pattern. If π turns out to be the length of the period of the pattern, the pattern analysis is done and we can proceed with the text analysis. Otherwise, the smallest witness found is good also for all the indices of the form $k\pi + 1$ which are in the first k_i–block, and we can proceed with the duels as in 2.

These three steps seem to require simultaneous write of different values but an implementation on the weaker common CRCW-PRAM is also possible [25]. In order to make our algorithm optimal, we take a more careful look at the algorithm described above. We redefine our block sizes k_i as follows,

$$k_0 = 1$$

$$k_i = \frac{m^{1-2^{-i}}}{\log\log m}, \text{ for } i = 1 \cdots \log\log m$$

$$k_i = 2k_{i-1}, \text{ for } i > \log\log m,$$

introducing $\log\log\log m$ more stages. Using this new sequence, $m/\log\log m$ processors are enough for step 1 of the original algorithm. Step 2 will now take $\log\log m$ time for the first **two** stages after which we will have less than $\sqrt{\frac{m}{\log\log m}}$ yet-to-be-computed witnesses. However, step 3 still needs m processors and we need to modify the entire algorithm.

We have two kinds of stages: non-periodic stages and periodic stages. Each kind is associated with certain initial conditions. The first stage is a non-periodic stage 1 for which the initial conditions hold vacuously because $k_0 = 1$ and no witnesses are computed.

A non-periodic stage i starts with at most one yet-to-be-computed witness in each k_{i-1}–block (in the first k_{i-1}–block the yet-to-be-computed witness is always the first). Moreover, all computed witnesses satisfy

$$\mathcal{W}[l] \le l + k_{i+1}. \tag{3.1}$$

A periodic stage i starts with some yet-to-be-computed witnesses in the first k_{i-1}–block. They are all the indices of the form $k\pi + 1$, where π is the period length of the first k_i–block. In a periodic stage i all computed witnesses satisfy

$$\mathcal{W}[l] \le l + k_i \tag{3.2}$$

and also,

$$\mathcal{W}[l] \le 2\pi \le k_i \quad \text{for } 2 \le l \le \pi. \tag{3.3}$$

In a non-periodic stage i we execute step 1 of our original algorithm and if all witnesses in the first k_i–block are computed we perform the duels of step 2, which result in at most one yet-to-be-computed witness in any k_i–block. The new witnesses in the first k_i–block obviously satisfy $\mathcal{W}[l] \le 2k_i \le k_{i+1}$. Hence, the new witnesses in the other k_i–blocks satisfy $\mathcal{W}[l] < l + k_{i+2}$. So all computed witnesses satisfy [1] with i increased by 1. If all witnesses in the first k_i–block have been computed we proceed in a non-periodic stage $i+1$; otherwise, we verify p to be the period length of the first k_{i+1}–block. If it is not, we found **the same** witness ($\le k_{i+1}$) for all the indices of the form $k\pi + 1$ in the first k_i–block and we continue with the duels of step 2 as in the previous case; otherwise we proceed with a periodic stage $i+1$. In both cases, the initial conditions obviously hold.

In a periodic stage i we first check if π is the period length of the first k_{i+1}–block. In case it is, we use the periodicity to compute witnesses for all indices l where $l \neq 1 \mod \pi$ in the first k_i–block as follows. Let $j = \lfloor \frac{l-1}{\pi} \rfloor \pi$. Set $\mathcal{W}[l] = j + \mathcal{W}[l-j] \le 2k_i \le k_{i+1}$ (by Equation 3.3). We then proceed with a periodic stage $i+1$, and the initial conditions obviously hold. Actually, Equation 3.3 might not hold immediately. By Equation 3.2 we have $\mathcal{W}[l] < k_{i+1}$ for $2 \le l \le \pi$. Since π is the period length of the first k_{i+1}–block, we can modify the witnesses to satisfy Equation 3.3 by Lemma 3.6.

If we find that π is not the period length of the first k_{i+1}–block, we actually find at once a witness for all indices of the form $kp+1$ in the first k_{i-1}–block. This witness is not larger than k_{i+1}. We then perform the duels in each of the k_{i-1}–blocks, which result in all computed witnesses satisfying Equation 3.1 and with at most one yet-to-be-computed witness in each k_{i-1}–block. These are the initial conditions for a non-periodic stage i. We then proceed with a non-periodic stage i. Note that unlike the non-optimal algorithm, we perform duels only if the next stage is non-periodic.

We now take a careful look at the last stage. Let r be maximal index such that $k_r < m$ and define $k_{r+1} = m$. As we have shown, duels can be made for all i and j where $i < j < \lceil m/2 \rceil + 1$, thus in a non-periodic stage r everything works well if we perform duels only in the first half of the pattern. In a periodic stage r we either verify the period of the whole pattern, or we find a witness and enter a non-periodic stage r.

Since we can be in a periodic stage i and a non-periodic stage i at most once for each i, the total number of operations is $O(m)$ and by Theorem 1.11 our algorithm is optimal.

Lemma 3.9 *The pattern processing phase can be implemented in $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors and in $O(\lceil \frac{n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ time on p processors.*

Proof: We have seen above that the pattern processing phase can be implemented in $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors. As in Lemma 3.8 It remains to show that if the number of available processors is p, then the algorithm takes $O(\lceil \frac{n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ time.

If $p < \frac{n}{\log\log n}$, then by Theorem 1.11 the algorithm described above can be slowed down to work on p processors in $O(\frac{n}{p})$ time. If $\frac{n}{\log\log n} \le p \le n$ then the bound above is still $O(\log\log n)$. If the number of processors if larger than n^2, then the simple algorithm of Lemma 3.2 solves the problem in constant time. The only remaining case is when $n \le p \le n^2$.

At stage number i, $k_i = p^{1-2^{-i}}$. The computation terminates after i steps when $p^{1-2^{-i}} \ge \frac{m}{2}$. By lemma 3.7, $\log\log_{\frac{2p}{m}} p$ steps are required. □

Theorem 3.10 *The string matching problem can be solved in $O(\log\log n)$ time on $\frac{n}{\log\log n}$ processors and in $O(\lceil \frac{n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ time on p processors.*

Proof: The proof follows from Lemma 3.8 and Lemma 3.9. □

3.3 An $\Omega(\log\log n)$ Lower Bound

In this section we describe a lower bound for a model that is similar to Valiant's parallel comparison tree model [86]. We assume the only access the algorithm has to the input strings is by comparisons that check whether two symbols are equal or not. The algorithm is allowed m comparisons in each round, after which it can proceed to the next round or terminate with the answer. We give a lower bound on the minimum number of rounds necessary in the worst case.

Consider a CRCW-PRAM that solves the string matching problem over a general alphabet. In this case the PRAM can perform comparisons, but not computation, with its input symbols. Thus, its execution can be partitioned into comparison rounds followed by computation rounds. Therefore, a lower bound for the number of rounds in the parallel comparison model immediately translates into a lower bound for the time of the CRCW-PRAM. If the pattern is given in advance and any preprocessing is free, then this lower bound does not hold, as Vishkin's [88] $O(\log^* m)$ algorithm shows. The lower bound also does not hold for CRCW-PRAM over a fixed alphabet strings, but no better algorithm is known at the moment. Similarly, finding the maximum in the parallel decision tree model has exactly the same lower bound [86], but for small integers the maximum can be found in constant time on a CRCW-PRAM as shown in Theorem 1.14.

We start by proving a lower bound for a related problem of finding the period length of a string. Given a string $\mathcal{S}[1..m]$ we prove that $\Omega(\log\log m)$ rounds are necessary for determining whether it has a period length smaller than $\frac{m}{2}$. Later we show how this lower bound translates into a lower bound for string matching.

3.3.1 A Lower Bound for Computing the Period Length

We show a strategy for an adversary to answer $\frac{1}{4}\log\log m$ rounds of comparisons after which it still has the choice of fixing the input string $\mathcal{S}[1..m]$ in two ways: in one the resulting string has a period of length smaller than $\frac{m}{2}$ and in the other it does not have any such period. This implies that any algorithm that terminates in less rounds can be fooled.

We say that an integer k is a possible period length if we can fix $\mathcal{S}$ consistently with answers to previous comparisons in such a way that k is a period length of $\mathcal{S}$. For such k to be a period length we need each residue class modulo k to be fixed to the same symbol, thus if $l \equiv j \mod k$, then $\mathcal{S}[l] = \mathcal{S}[j]$.

At the beginning of round i the adversary will maintain an integer k_i that is a possible period length. The adversary answers the comparisons of round i in such a way that some k_{i+1} is a possible period length and few symbols of $\mathcal{S}$ are fixed. Let $K_i = m^{1-4^{-(i-1)}}$. The adversary will maintain the following invariants that hold at the beginning of round number i:

1. k_i satisfies $\frac{1}{2}K_i \le k_i \le K_i$.

2. If $\mathcal{S}[l]$ was fixed then for every $j \equiv l \mod k_i$ $\mathcal{S}[j]$ was fixed to the same symbol.

3. If a comparison was answered as equal then both symbols compared were fixed to the same value.

4. If a comparison was answered as unequal, then

a. it was between different residues modulo k_i;

b. if the symbols were fixed then they were fixed to different values.

5. The number of fixed symbols f_i satisfies $f_i \leq K_i$.

Note that invariants 3 and 4 imply consistency of the answers given so far. Invariants 2, 3 and 4 imply that k_i is a possible period length: if we fix all symbols in each unfixed residue class modulo k_i to a new symbol, a different symbol for different residue classes, we obtain a string consistent with the answers given so far that has a period length k_i.

We start at round number 1 with $k_1 = K_1 = 1$. It is easy to see that the invariants hold initially. We show how to answer the comparisons of round i and how to choose k_{i+1} so that the invariants still hold. All multiples of k_i in the range $\frac{1}{2}K_{i+1} \ldots K_{i+1}$ are candidates for the new k_{i+1}. A comparison $\mathcal{S}[l] = \mathcal{S}[j]$ must be answered as equal if $l \equiv j \mod k_{i+1}$. We say that k_{i+1} *forces* this comparison.

Theorem 3.11 *(Rosser and Schoenfeld [80]) For large enough*[3] *n, the number of primes between 1 and n denoted by $\pi(n)$ satisfies, $\frac{n}{\ln n} \leq \pi(n) \leq \frac{5}{4}\frac{n}{\ln n}$.*

Corollary 3.12 *The number of primes between $\frac{1}{2}n$ and n is greater than $\frac{1}{4}\frac{n}{\log n}$.*

Lemma 3.13 *If $p, q \geq \sqrt{\frac{m}{k_i}}$ and are relatively prime, then a comparison $\mathcal{S}[l] = \mathcal{S}[k]$ is forced by at most one of pk_i and qk_i.*

Proof: Assume $l \equiv k \mod pk_i$, $l \equiv k \mod qk_i$ for $1 \leq l, k \leq m$. Then also $l \equiv k \mod pqk_i$. But $pqk_i \geq m$ and $1 \leq l, k \leq m$ which implies $l = k$; a contradiction. □

Lemma 3.14 *The number of candidates for k_{i+1} that are prime multiples of k_i and satisfy $\frac{1}{2}K_{i+1} \leq k_{i+1} \leq K_{i+1}$ is greater than $\frac{K_{i+1}}{4K_i \log m}$. Each such candidate satisfies the condition of Lemma 3.13.*

Proof: These candidates are of the form pk_i for prime p. The number of such prime values of p can be estimated using the Corollary 3.12. It is at least

$$\frac{1}{4}\frac{K_{i+1}}{k_i \log \frac{K_{i+1}}{k_i}} \geq \frac{K_{i+1}}{4K_i \log m}.$$

Each one of these candidates also satisfies the condition of Lemma 3.13 since $k_i \leq K_i$, $pk_i \geq \frac{K_{i+1}}{2}$ and

$$p^2 \geq \frac{1}{k_i}\frac{K_{i+1}^2}{4K_i} = \frac{1}{k_i}\frac{m^{2-2\cdot 4^{-i}}}{4m^{1-4^{-(i-1)}}} = \frac{m}{k_i}\frac{1}{4}m^{2\cdot 4^{-i}} \geq \frac{m}{k_i}.$$

□

[3] The theorem hold for $n \geq 115$.

Lemma 3.15 *There exists a candidate for k_{i+1} in the range $\frac{1}{2}K_{i+1} \ldots K_{i+1}$ that forces at most $\frac{4mK_i \log m}{K_{i+1}}$ comparisons.*

Proof: By Lemma 3.14 there are at least $\frac{K_{i+1}}{4K_i \log m}$ such candidates that are prime multiples of k_i and satisfy the condition of Lemma 3.13. By Lemma 3.13 each of the m comparisons is forced by at most one of them. So the total number of comparisons forced by all these candidates is at most m. Thus, there is a candidate that forces at most $\frac{4mK_i \log m}{K_{i+1}}$ comparisons. □

Lemma 3.16 *For m large enough and $i \leq \frac{1}{4}\log\log m$, $1 + m^{2 \cdot 4^{-i}} 16 \log m \leq m^{3 \cdot 4^{-i}}$.*

Proof: For m large enough,

$$\log\log(1 + 16\log m) < \frac{1}{2}\log\log m = (1 - \frac{2}{4})\log\log m$$

$$\log(1 + 16\log m) < 4^{-\frac{1}{4}\log\log m} \log m$$

$$1 + 16\log m < m^{4^{-\frac{1}{4}\log\log m}} \leq m^{4^{-i}},$$

from which the lemma follows. □

Lemma 3.17 *Assume the invariants hold at the beginning of round i and the adversary chooses k_{i+1} to be a candidate that forces at most $\frac{4mK_i \log m}{K_{i+1}}$ comparisons. Then the adversary can answer the comparisons in round i so that the invariants also hold at the beginning of round $i+1$.*

Proof: By Lemma 3.15 such k_{i+1} exists. For each comparison that is forced by k_{i+1} and is of the form $\mathcal{S}[l] = \mathcal{S}[j]$ where $l \equiv j \mod k_{i+1}$ the adversary fixes the residue class modulo k_{i+1} to the same new symbol (a different symbol for different residue classes). The adversary answers comparisons between fixed symbols based on the value they are fixed to. All other comparisons involve two positions in different residue classes modulo k_{i+1} (and at least one unfixed symbol) and are always answered as unequal.

Since k_{i+1} is a multiple of k_i, the residue classes modulo k_i split; each class splits into $\frac{k_{i+1}}{k_i}$ residue classes modulo k_{i+1}. Note that if two indices are in different residue classes modulo k_i, then they are also in different residue classes modulo k_{i+1}; if two indices are in the same residue class modulo k_{i+1}, then they are also in the same residue class modulo k_i.

We show that the invariants still hold.

1. The candidate we chose for k_{i+1} was in the required range.

2. Residue classes that were fixed before this round now split into several residue classes, all are fixed. Any symbol fixed at this round causes its entire residue class modulo k_{i+1} to be fixed to the same symbol.

3. Equal answers of previous rounds are not affected since the symbols involved were fixed to the same value by the invariants held before. Equal answers of this round are either between symbols that were fixed before to the same value or are within the same residue class modulo k_{i+1} and the entire residue class is fixed to the same value.

4. **a.** Unequal answers of previous rounds are between different residue classes modulo k_{i+1} since residue classes modulo k_i split. Unequal answers of this round are between different residue classes because comparisons within the same residue class modulo k_{i+1} are always answered as equal.

   b. Unequal answers that involve symbols which were fixed before this round are consistent because fixed values dictate the answers to the comparisons. Unequal answers that involve symbols that are fixed at the end of this round and at least one was fixed at this round are consistent since a new symbol is used for each residue class fixed.

5. We prove inductively that $f_{i+1} \leq K_{i+1}$. We fix at most $\frac{4mK_i \log m}{K_{i+1}}$ residue classes modulo k_{i+1}. There are k_{i+1} such classes and each class has at most $\lceil \frac{m}{k_{i+1}} \rceil \leq \frac{2m}{k_{i+1}}$ elements. By Lemma 3.16 and simple algebra the number of fixed elements satisfies:

$$
\begin{aligned}
f_{i+1} &\leq f_i + \frac{2m}{k_{i+1}} \frac{4mK_i \log m}{K_{i+1}} \\
&\leq K_i \left[1 + \left(\frac{m}{K_{i+1}} \right)^2 16 \log m \right] \\
&\leq m^{1-4^{-(i-1)}} (1 + m^{2 \cdot 4^{-i}} 16 \log m) \\
&\leq m^{1-4^{-i}} = K_{i+1}.
\end{aligned}
$$

□

Theorem 3.18 *Any comparison-based parallel algorithm that finds the period length of a string $\mathcal{S}[1..m]$ using m comparisons in each round requires at least $\frac{1}{4} \log \log m$ rounds. If p comparisons are allowed at each round, then $\Omega(\lceil \frac{n}{p} \rceil + \log \log_{\lceil 1+p/m \rceil} 2p)$ rounds are required.*

Proof: Fix an algorithm that finds the period of $\mathcal{S}$ and let the adversary described above answer the comparisons. After $i = \frac{1}{4} \log \log m$ rounds $f_{i+1}, k_{i+1} \leq m^{1-4^{-\frac{1}{4} \log \log m}} = \frac{m}{2\sqrt{\log m}} \leq \frac{m}{2}$. The adversary can still fix $\mathcal{S}$ to have a period length k_{i+1} by fixing each remaining residue class modulo k_{i+1} to the same symbol, different symbol for each class. Alternatively, the adversary can fix all unfixed symbols to different symbols. Note that this choice is consistent with all the the comparisons answered so far by invariants 3 and

4, and the string does not have any period length smaller than $\frac{m}{2}$. Consequently, any algorithm that terminates in less than $\frac{1}{4}\log\log m$ rounds can be fooled.

It remains to show that if the p comparisons are allowed at each round, then the algorithm takes $\Omega(\lceil\frac{m}{p}\rceil + \log\log_{\lceil 1+p/m\rceil} 2p)$ rounds. If $p < \frac{m}{\log\log m}$, then $\Omega(\frac{m}{p})$ rounds are required. If $\frac{m}{\log\log m} \leq p \leq m$ then the bound above is still applies. If $p > m^2$, then the simple algorithm of Lemma 3.2 solves the problem in constant time. The only remaining case is when $m \leq p \leq m^2$.

We change m to p in the appropriate places of the lower bound proof above. In particular we choose $K_i = p^{1-4^{-(i-1)}}$. The adversary can go on as long as $K_i \leq \frac{m}{2}$. By Lemma 3.7, $\Omega(\log\log_{\frac{2p}{m}} p)$ rounds are necessary.

By combining all these bounds into a single expression we get the promised bound of $\Omega(\lceil\frac{m}{p}\rceil + \log\log_{\lceil 1+p/m\rceil} 2p)$ rounds. □

Theorem 3.19 *The lower bound also holds for order comparisons.*

Proof: The adversary gradually defines the linear order of the symbols. He does it in such a way that the answers to comparisons in round i are determined at the round or before. The order is determined by a lexicographic order on a name given to each symbol and is extended for unfixed symbols at each round.

At round i, after k_{i+1} is chosen and before fixing new symbols, all names of unfixed symbols $\mathcal{S}[l]$ are extended to be $< l \bmod k_2, l \bmod k_3, ..., l \bmod k_{i+1} >$. If a symbol $\mathcal{S}[l]$ was fixed at round $j \leq i$ its entire residue class is assigned the the name $< l \bmod k_2, l \bmod k_3, ..., l \bmod k_{j+1} >$ which is never changed. Note that all other names cannot have this name as a prefix. If two symbols compared at round i are in the same residue class modulo k_{i+1}, then they were fixed at this round or before and given the same name. Symbols that are compared at round i and are in different residue classes modulo k_{i+1} have different name unless they were fixed earlier to the same value. The result of the comparison ($<, =, >$) is already determined even if the full final name of the symbols is not defined yet. At the end, if the residue classes modulo k_{i+1} are fixed in such a way that k_{i+1} is the period length, then the full names are given as before. If all symbols are fixed to be different, the name given to unfixed position l is $< l \bmod k_2, l \bmod k_3, ..., l \bmod k_{i+1}, l >$. □

3.3.2 A Lower Bound for String Matching

The next theorem shows directly that the lower bound of the previous section also holds for the string matching problem. The same bound for string matching can be implied by the algorithm for finding all periods of a string in section 3.4.

Theorem 3.20 *The lower bound holds also for any comparison-based string matching algorithm when $n = O(m)$.*

Proof: Fix a string matching algorithm. We present to the algorithm a pattern $\mathcal{P}[1..m]$ which is $\mathcal{S}[1..m]$ and a text $\mathcal{T}[1..2m-1]$ which is $\mathcal{S}[2..2m]$, where $\mathcal{S}$ is a string of length $2m$ generated by the adversary in the way described above. (We use the same adversary that we used in the previous proof; the adversary sees all comparisons as comparisons between symbols in $\mathcal{S}$.) After $\frac{1}{4}\log\log 2m$ rounds the adversary still has the choice of fixing $\mathcal{S}$ to have a period length smaller than m, in which case we will have an occurrence of $\mathcal{P}$ in $\mathcal{T}$, or to fix all unfixed symbols to completely different characters, which implies that there would be no such occurrence. Thus, the lower bound holds also for any such string matching algorithm. □

3.4 Finding All Periods

In this section we present an $O(\log\log n)$ time $\frac{n}{\log\log n}$-processor CRCW-PRAM algorithm to compute all periods of a string [9, 27]. If the number of available processors is p we show that the algorithm takes $O(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time.

The period of a string is computed in linear time as a step in Knuth, Morris and Pratt's sequential string matching algorithm [66] and in optimal $O(\log\log n)$ parallel time on a CRCW-PRAM as shown in Section 3.2.4. The parallel algorithm in Section 3.2.4 computes the period $\pi_1^{\mathcal{S}}$ only if $\pi_1^{\mathcal{S}} \leq \lceil\frac{n}{2}\rceil$; knowing the fact that $\pi_1^{\mathcal{S}} > \lceil\frac{n}{2}\rceil$ is sufficient to obtain a good string matching algorithm. We show that given an optimal parallel algorithm for string matching, one can compute all 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[4] the algorithm of section 3.2 can be used to obtain an optimal $O(\log\log n)$ time CRCW-PRAM algorithm that computes the exact period of a string.

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

We will prove the following theorem:

Theorem 3.21 *There exists an algorithm to compute $P[1..n]$ that takes $O(\log\log n)$ time using $\frac{n}{\log\log n}$ processors. If p processors are available the algorithm takes $O(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time.*

Corollary 3.22 *The exact period of a string S can be computed in the same time and processor bounds of Theorem 3.21.*

[4]The algorithm described in this paper uses a string matching procedure as a "black-box". Using the string matching algorithm described above would result in an optimal $O(\log\log n)$ time algorithm that is the best possible in the case of a general alphabet as implied by a the lower bound above. It is unknown if a faster algorithm exists for string matching in the case of a fixed alphabet. If such algorithm exists it would immediately imply a faster algorithm for finding the periods. The same holds for the algorithms for squares and palindromes described in the following chapters.

Proof: The period of S is the smallest i such that $P[i]$ is $true$. By Theorem 1.14 given the array $P[i]$ we can find the period in constant time using an n-processor CRCW-PRAM. By Theorem 1.11 this step can be slowed down to work in optimal $O(\log\log n)$ time or in $O(\frac{n}{p})$ time on p processors. $\Box$

Proof of Theorem 3.21: The algorithm will proceed in independent stages which are all computed simultaneously and described in the next section. In stage number η, $0 \le \eta < m$, 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 the parallel string matching algorithm of Section 3.2.

Since all stages of our algorithm are executed in parallel the total number of operation performed in all stages is $\sum_\eta O_\eta \le \sum_\eta \left(\frac{2}{3}\right)^\eta n = O(n)$ and the time is $\max T_\eta = O(\log\log n)$. By Theorem 2.1 the algorithm can be implemented using $\frac{n}{\log\log n}$ processors in $O(\log\log n)$ time.

It remains to show that if the number of available processors is p the algorithm takes $O(\lceil \frac{n}{p} \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ time. If $p < \frac{n}{\log\log n}$ then by Theorem 2.1 the algorithm can be slowed down to work in $O(\frac{n}{p})$ time. If $\frac{n}{\log\log n} \le p \le n$ then the bound above is still $O(\log\log n)$. If $p > n$, then stage number η can be implemented in $T_\eta = O(\log\log_{\lceil 1+p/n \rceil} \frac{2p}{n} l_\eta)$ time using $\frac{p}{n} l_\eta$ processors. The total number of processors used for all stages is $\sum_\eta \frac{p}{n} l_\eta \le \frac{p}{n} \sum_\eta \left(\frac{2}{3}\right)^\eta n = O(p)$ and the time is $\max T_\eta = O(\log\log_{\lceil 1+p/n \rceil} 2p)$. $\Box$

A Single Stage

We now describe a single stage η, $0 \le \eta < m$, 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 $\mathcal{S}[0..n-p] = \mathcal{S}[p..n]$, there must be an occurrences of $\mathcal{S}[0..l_{\eta+1}]$ starting at each position p which is a period of $\mathcal{S}$ in the range computed by this stage.

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

If there were no occurrences found, the string $\mathcal{S}$ 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 $\mathcal{S}[0..l_{\eta+1}]$ in $\mathcal{S}[0..l_\eta - 1]$. 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 $\mathcal{S}[0..l_{\eta+1}]$ in $\mathcal{S}[n - l_\eta + 1..n]$, it can be verified to be a period in $O(l_\eta)$ operations. However, if there are $r > 1$ occurrences, $O(rl_\eta)$ 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 $\mathcal{S}$.

Lemma 3.23 *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 1.6 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 3.24 *The sequences $\{p_i\}$ and $\{q_i\}$ form an arithmetic progression with difference $\mathcal{P}$, where $\mathcal{P}$ is the period of $\mathcal{S}[0..l_{\eta+1}]$.*

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 3.23 the p_i's and q_i's form an arithmetic progression with a difference $\mathcal{P}$, the period of $\mathcal{S}[0..l_{\eta+1}]$. □

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 $\mathcal{S}$ immediately as we show in the following lemma.

Lemma 3.25 *If $k < r$ then q_i is not a period of $\mathcal{S}$ for $1 \leq i \leq r - k$.*

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

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

1. If $\mathcal{S}[q_r + \mathcal{P}..\mathcal{N}] \neq \mathcal{S}[0..\mathcal{N} - q_r - \mathcal{P}]$, 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 3.26 *If $\mathcal{S}[q_r + \mathcal{P}..\mathcal{N}] \neq \mathcal{S}[0..\mathcal{N} - q_r - \mathcal{P}]$ then, $\mathcal{S}$ 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 3.25 all q_i, $1 \le i < r-k+1$, are not periods. Assume q_i is a period and $i > r-k+1$, then $\mathcal{S}[q_i..n] = \mathcal{S}[0..n-q_i]$. Since $r-i+2 \le k$ and $p_j = (j-1)\mathcal{P}$, also $\mathcal{S}[q_r+\mathcal{P}..\mathcal{N}] = \mathcal{S}[p_{r-i+2}..\mathcal{N}-q_i]$. By the assumption of a mismatch $\mathcal{S}[q_r+\mathcal{P}..\mathcal{N}] \ne \mathcal{S}[0..\mathcal{N}-q_r-\mathcal{P}]$. So $\mathcal{S}[p_{r-i+2}..\mathcal{N}-q_i] \ne \mathcal{S}[0..\mathcal{N}-q_r-\mathcal{P}]$. But $\mathcal{S}[p_{r-i+2}..p_{r-i+2}+l_{\eta+1}] = \mathcal{S}[0..l_{\eta+1}]$ and also $\mathcal{N}-q_r-\mathcal{P} \le l_{\eta+1}$; contradiction. $\Box$

Lemma 3.27 *If* $\mathcal{S}[q_r+\mathcal{P}..n] = \mathcal{S}[0..n-q_r-\mathcal{P}]$ *then:*

a. *If* $r < k$ *then* $q_1, .., q_r$ *are periods of* $\mathcal{S}$.

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

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

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

$$\mathcal{S}[0..p_{r-i+1}+l_{\eta+1}] = \mathcal{S}[q_i..q_r+l_{\eta+1}] \tag{3.4}$$

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

$$\mathcal{S}[p_{r-i+2}..p_{r-i+2}+l_{\eta+1}] = \mathcal{S}[0..l_{\eta+1}]. \tag{3.5}$$

But $n-q_r-\mathcal{P} < l_{\eta+1}$ and $\mathcal{S}[q_r+\mathcal{P}..n] = \mathcal{S}[0..n-q_r-\mathcal{P}]$. By taking prefixes of Equation 3.5

$$\mathcal{S}[q_r+\mathcal{P}..n] = \mathcal{S}[p_{r-i+2}..p_{r-i+2}+n-q_r-\mathcal{P}]. \tag{3.6}$$

By combining equalities 3.4 and 3.6 we get that $\mathcal{S}[0..n-q_i] = \mathcal{S}[q_i..n]$. $\Box$

The computation in stage η can be summarized as follows:

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

Theorem 3.28 *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 Lemma 3.25, Lemma 3.26 and Lemma 3.27. 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 the string matching algorithm of Section 3.2. 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 assume 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. □

Lemma 3.29 *Stage number η can be implemented in $O(\log\log_{\lceil 1+p/n\rceil} \frac{2p}{n} l_\eta)$ time on $\frac{p}{n} l_\eta$ processors if $p \geq n$.*

Proof: The calls to the string matching algorithm take $O(\log\log_{\lceil 1+p/n\rceil} \frac{2p}{n} l_\eta)$ time if $p \geq n$ and $\frac{p}{n} l_\eta$ processors are available for stage number η. The rest of the work can be done in constant time since the number of processors is larger than l_η. □

3.5 Open Problems

There are still many interesting open question about parallel string matching. The most important in our opinion are:

1. String matching over a fixed alphabet. The lower bound of Section 3.3 assumes the input strings are drawn from a general or ordered alphabet and the only access to them is by comparisons. The lower and upper bounds for the string matching problem over a general alphabet are identical to those for comparison based maximum finding algorithm obtained by Valiant [86]. A constant time algorithm can find the maximum of integers in a restricted range [42] what suggests the possibility of a faster string matching algorithm.

2. String matching with long text strings. If the text string is much longer than the pattern, the lower bound of Section 3.3 does not apply. Indeed, on a comparison model where all computation is free one can do the preprocessing for Vishkin's fast algorithm in constant time using m^2 processors. If $n = m^2$ the n processors are available to preprocess the short pattern. However, it is not known if the preprocessing can be performed on a CRCW-PRAM, or how is can be done faster with less then m^2 processors on a comparison model.

3. String matching on the CREW-PRAM and the EREW-PRAM models. The fastest deterministic optimal CREW-PRAM string matching algorithm is obtained by slowing down the CRCW algorithm which was described in this chapter to work in $O(\log m \log\log m)$ time. What is the exact bound on the CREW and the EREW models?

Chapter 4

Parallel Algorithms for Detecting Squares

Squares and strings that contain squares have important implications in coding theory, automata theory, formal language theory, data compression and recently also in molecular biology. Squares are used implicitly in many string algorithm since the basic periodicity lemma (Lemma 1.6) can be easily applied when a substring is repeated at least twice.

A string $\mathcal{S}[1..n]$ is *square-free* if it does not contain any substring that is a square. In this chapter we describe optimal parallel algorithms and lower bounds for testing if a string is square-free on the CRCW-PRAM model. We obtain bounds of $\Theta(\log\log n)$ time on $\frac{n\log n}{\log\log n}$ processors or $\Theta(\lceil\frac{n\log n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time on p processors.

4.1 Historical Overview

There exist few sequential algorithms that test if a string is square-free. Algorithms by Apostolico and Preparata [12], by Crochemore [35], Rabin [77] and by Main and Lorentz [72] find all the squares in a string of length n in $O(n\log n)$ time. Main and Lorentz [72] 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 [73] show that the latter problem of deciding whether a string is square-free can be solved in $O(n)$ time on a constant alphabet. Crochemore [36] also gave a linear time algorithm for this problem.

In parallel, an algorithm by Crochemore and Rytter [39] 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 [7] 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. Apostolico's [7] 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 [7] 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 [7] algorithm for testing square-freeness in the case of a finite alphabet is also optimal since the time-processor product is $O(n)$, the best running time of a sequential algorithm for this problem.

In this chapter we develope an optimal parallel algorithm and a lower bound for testing if a string is square-free. We give tight bounds for solving the problem on a general alphabet for any number of processors used. The algorithm described in this chapter is a parallel version of the sequential algorithm of Main and Lorentz [73].

4.2 An $O(\log \log n)$ Time Algorithm

In this section we describe optimal parallel algorithm for testing if a string is square-free on the CRCW-PRAM model. We obtain bounds of $O(\log \log n)$ time on $\frac{n \log n}{\log \log n}$ processors or $O(\lceil \frac{n \log n}{p} \rceil + \log \log_{\lceil 1+p/n \rceil} 2p)$ time on p processors.

We will prove the following theorem:

Theorem 4.1 *There exists an algorithm that tests if a string $\mathcal{S}[1..n]$ is square-free and takes $O(\log \log n)$ time on $\frac{n \log n}{\log \log n}$ processors or $O(\lceil \frac{n \log n}{p} \rceil + \log \log_{\lceil 1+p/n \rceil} 2p)$ time on p processors.*

Proof: 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 the string matching algorithm in Section 3.2. Since there are $O(\log n)$ stages, the total number of operations is $O(n \log n)$. By Theorem 1.11 the complete algorithm can be implemented in $\max T_\eta = O(\log \log n)$ time and $\frac{n \log n}{\log \log n}$ processors.

It remains to show that if p processors are available in each round then the algorithm takes $O(\lceil \frac{n \log n}{p} \rceil + \log \log_{\lceil 1+p/n \rceil} 2p)$ time. If $p < \frac{n \log n}{\log \log n}$ then by Theorem 1.11 the algorithm above can be slowed down to work in $\frac{n \log n}{p}$ time. If $\frac{n \log n}{\log \log n} \leq p \leq n \log n$ then the bound above is still $O(\log \log n)$. If $p > n^2$ then the problem can be solved in constant time by a naive algorithm similar to the algorithm in Lemma 3.2. The interesting case is where $n \log n \leq p \leq n^2$. In this case stage η can be implemented in $T_\eta = O(\log \log_{\lceil 1+p/n \rceil} 2p)$ using $\frac{p}{\log n}$ processors. The total number of processors used for all stages is p and the time is $\max T_\eta = O(\log \log_{\lceil 1+p/n \rceil} 2p)$. $\square$

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 $\mathcal{S}[q]$ out of the range $\mathcal{S}[1..n]$ is assumed to be answered as unequal.

Partition the input string $\mathcal{S}[1..n]$ 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 $\mathcal{S}[(k-1)l_\eta + 1..kl_\eta]$. 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, $\mathcal{S}[(k-1)l_\eta + 1..kl_\eta]$, in $\mathcal{S}[(k+1)l_\eta..(k+4)l_\eta - 2]$. 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 $\mathcal{S}[(k-1)l_\eta + 1..kl_\eta]$ there must be an occurrence of $\mathcal{S}[(k-1)l_\eta + 1..kl_\eta]$ at position $(k-1)l_\eta + |x| + 1$. This occurrence is included in the $\{p_i\}$ sequence.

Lemma 4.2 *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 $\mathcal{S}[(k-1)l_\eta + 1..kl_\eta]$.*

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 $\mathcal{S}[\zeta - l] = \mathcal{S}[\zeta]$ and if $\mathcal{S}[\zeta] = \mathcal{S}[\zeta + l]$. Let ζ_L be the largest index in this range such that $\mathcal{S}[kl_\eta + 1..\zeta_L] = \mathcal{S}[kl_\eta + l + 1..\zeta_L + l]$ and ζ_R be the smallest index such that $\mathcal{S}[\zeta_R..(k-1)l_\eta + l] = \mathcal{S}[\zeta_R - l..(k-1)l_\eta]$. Using the algorithm of Fich, Ragde and Wigderson [42] we can find ζ_L and ζ_R in constant time and $O(l_\eta)$ operations. $\square$

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

If $\zeta_L \geq \zeta_R - 1$ then $\mathcal{S}[\zeta_R - l..\zeta_L] = \mathcal{S}[\zeta_R..\zeta_L + l]$. 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 $\mathcal{S}[(k-1)l_\eta + 1..kl_\eta]$ that starts at position π, then for all ζ in the range $\pi + |x| \leq \zeta \leq (k-1)l_\eta + l$, $\mathcal{S}[\zeta - |x|] = \mathcal{S}[\zeta]$ and for all ζ in the range $kl_\eta < \zeta < \pi + |x|$, $\mathcal{S}[\zeta] = \mathcal{S}[\zeta + |x|]$. But $\zeta_L < \zeta_R - 1$ so $\mathcal{S}[\zeta_L + 1] \neq \mathcal{S}[\zeta_L + |x| + 1]$ and $\mathcal{S}[\zeta_R - |x| - 1] \neq \mathcal{S}[\zeta_R - 1]$; a contradiction to the existence of a square of length $|x|$ at position π. $\square$

We could verify all the p_i's in constant time using Lemma 4.2 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 4.3 *If the number of occurrences of* $\mathcal{S}[(k-1)l_\eta+1..kl_\eta]$ *in* $\mathcal{S}[(k+1)l_\eta..(k+4)l_\eta-2]$ *is larger than two, then* $\mathcal{S}[1..n]$ *has a square. This square is actually shorter than the squares which are supposed to be found by this stage.*

Proof: All the occurrences $\{p_i\}$, $1 \le i \le r$ satisfy $(k+1)l_\eta \le p_i \le (k+3)l_\eta - 1$. If $r \ge 3$ then either $p_2 - p_1 \le l_\eta$ or $p_3 - p_2 \le l_\eta$. 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). $\square$

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

1. Compute the $\{p_i\}$ sequence.
2. If the $\{p_i\}$ sequence has more than two elements then by Lemma 4.3 the string $\mathcal{S}[1..n]$ 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 4.2.

Lemma 4.4 *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 $\mathcal{S}[1..n]$ has a any square xx, $2^\eta - 1 \le |x| < 2^{\eta+1} - 1$, then some square will be found.

Assume there is such a square. Since $2l_\eta - 1 \le |x|$, 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 4.3. 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 $\frac{n}{l_\eta}l_\eta = O(n)$. $\square$

Lemma 4.5 *Stage number* η *can be implemented in* $O(\log\log_{\lceil 1+p/n\rceil} \frac{2p}{n}l_\eta)$ *time on* $\frac{p}{\log n}$ *processors if* $\frac{p}{\log n} \ge n$.

Proof: Stage η consists of $\frac{n}{l_\eta}$ independent sub-stages. In each sub-stage, step number 1 takes $O(\log\log_{\lceil 1+p/n\rceil} \frac{2p}{n}l_\eta)$ time and $O(\frac{p}{n\log n}l_\eta)$ processors. 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 $O(\log\log_{\lceil 1+p/n\rceil} \frac{2p}{n}l_\eta)$ and the total number of processors used is $\frac{pl_\eta}{n\log n}\frac{n}{l_\eta} = O(\frac{p}{logn})$. $\square$

Theorem 4.1 follows from the last two lemmas since the $\log n$ stages are executed in parallel.

4.3 An $\Omega(\log\log n)$ 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 in Section 3.3. 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.

The lower bound in Section 3.3. 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 repeat the complete details of that lower bound. We only use the fact that the adversary of Section 3.3 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 $\mathcal{S}[1..n]$ 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 $\mathcal{S}[i] = \mathcal{S}[j]$, for $i < j$, then for any integer k, such that $k \equiv i \, mod \, (j-i)$ and $1 \leq k \leq n$, $\mathcal{S}[k] = \mathcal{S}[i]$.

Lemma 4.6 *The string generated by the adversary of Section 3.3 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 $\mathcal{S}[i+k] = \mathcal{S}[i+l+k]$ for $k = 0..l-1$. By the property of the string generated by the adversary, the string $\mathcal{S}[1..n]$ has a period l, which is smaller than half the length of $\mathcal{S}$. □

Now, we are ready to prove the lower bound.

Theorem 4.7 *Any optimal parallel algorithm that tests if a string $\mathcal{S}[1..n]$ is square-free using up to $n\log n$ comparisons in each round takes $\Omega(\log\log n)$ time. If p comparisons are allowed then the bound becomes $\Omega(\lceil\frac{n\log n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ time on p processors.*

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 [72]. Assume an algorithm performs $p = n\log n$ comparisons in each round. By the lower bound of Section 3.3 and Lemma 4.6 after $\Omega(\log\log n)$ rounds the adversary can decide if $\mathcal{S}[1..n]$ is square-free or not, fooling any algorithm which terminates in less rounds. □

4.4 Open Problems

There are few open questions about squares in sequential and parallel computation:

- The lower bound of Main and Lorentz [72] is for a general alphabet and it does not hold when order comparisons are allowed. It is unknown if a linear time sequential algorithm to test for squares is possible under the ordered alphabet assumption.

- The only algorithm for squares that works over a general alphabet is the off-line algorithm of Main and Lorentz [72]. It is unknown if an on-line algorithm exists over a general alphabet.

Chapter 5

Parallel Algorithms for Finding Palindromes

Palindromes have been studied for centuries as word puzzles and have some uses in formal language theory. Although the definitions of squares and palindromes are very similar, there is a big computational difference between these two terms. An example from formal language is that the language of all strings that contain palindromes over a given alphabet is easily shown to be *context-free* [57], while the language of all strings that contain squares is not *context-free* as shown by Ross and Winklmann [79].

We show how the algorithm that finds all periods of a string from Section 3.4 can be used also to find all initial palindromes in the same time and processor bounds. We prove also a matching lower bound for this problem under the assumption of a general alphabet.

5.1 Historical Overview

Manacher [74] presented an on-line algorithm that finds the smallest initial palindromes of a string. This algorithm can be modified easily to find all initial palindromes and even all other palindromes in a string. Galil [45] improved Manacher's algorithm to a real-time algorithm.

Knuth,Morris and Pratt [66] consider the problem of recognizing strings in *Palstar*, that is defined as P^*, when P is the set of all even palindromes (the concatenation of all even length palindromes). They give a linear time algorithm for this problem. This algorithm does not extend to recognize strings in a different version of Palstar that includes concatenations of all non-trivial palindromes (length two or more, odd and even). Galil and Seiferas [50] discovered an on-line algorithm that can recognize strings in both versions of Palstar in linear time.

In parallel computation, Crochemore and Perrin [40] consider the problems of finding all palindromes in a string and recognizing strings in Palstar and give $O(\log n)$ time $\frac{n}{\log n}$-processor algorithms.

5.2 An $O(\log \log n)$ Time Algorithm

Theorem 5.1 *All initial palindromes of a string $\mathcal{S}$ can be computed in $O(\log \log n)$ time on $\frac{n}{\log \log n}$ processors or in $\Theta(\lceil \frac{n}{p} \rceil + \log \log_{\lceil 1+p/n \rceil} 2p)$ time on p processors.*

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 [43]. See Figure 5.1 □

```
a b a a b $ b a a b a
                a b a a b $ b a a b a
```

(In the figure, the overlapping part **a b a** is shown in bold in both rows.)

Figure 5.1: The string '*abaab*' has an initial palindrome '*aba*'. This initial palindrome corresponds to the period '*abaab\$ba*' of the string '*abaab\$baaba*'.

5.3 An $\Omega(\log \log n)$ Lower Bound

Given a string $\mathcal{S}[0..n]$, we say that it has an initial palindrome of length k if $\mathcal{S}[i] = \mathcal{S}[k-i-1]$ for $i = 0, ..., k-1$. We modify the lower bound of Section 3.3 to a lower bound for determining whether a string $\mathcal{S}[0..n]$ has a an initial palindrome of length larger than $\frac{n}{2}$. Since there are minor modification in the details of the lower bound we include the complete proofs.

We show a strategy for an adversary to answer $\frac{1}{4} \log \log n$ rounds of comparisons after which it still has the choice of fixing the input string $\mathcal{S}$ in two ways: in one the resulting string has an initial palindrome of length larger than $\frac{n}{2}$ and in the other it does not have any such initial palindrome. This implies that any algorithm that terminates in less rounds can be fooled.

We say that an integer k is a possible period length of $\mathcal{S}[0..n]$ if we can fix $\mathcal{S}$ consistently with answers to previous comparisons in such a way that k is a period of $\mathcal{S}$. For such k to be a period length we need each residue class modulo k to be fixed to the same symbol, thus if $l \equiv j \mod k$ then $\mathcal{S}[l] = \mathcal{S}[j]$.

We say that an integer k is a possible initial palindrome of $\mathcal{S}[0..n]$ if we can fix $\mathcal{S}$ consistently with answers to previous comparisons in such a way that $\mathcal{S}$ has an initial palindrome of length k. For such k to be an initial palindrome length we need that if $l \equiv k - j - 1$ then $\mathcal{S}[l] = \mathcal{S}[j]$.

For an integer k to be a possible period and a possible initial palindrome we need that both conditions hold. That is, if $l \equiv j \mod k$ or if $l \equiv -j-1 \mod k$ then $\mathcal{S}[l] = \mathcal{S}[j]$. We call such k a possible *palindromic-period length.*

Let $l = qk + r$ such that $0 \le r < k$, that is $r = l \mod k$. Define $\phi_k(l)$ as:

$$\phi_k(l) = \begin{cases} r & \text{if } r < \lceil \frac{k}{2} \rceil \\ k - r - 1 & \text{otherwise} \end{cases}$$

Using this notation, k is a palindromic-period length of $\mathcal{S}$ if for any two indices l and j that satisfy $\phi_k(l) = \phi_k(j)$ also $\mathcal{S}[l] = \mathcal{S}[j]$. If $l \equiv j \mod k$ we say that l and j are in the *same residue class* modulo k. If $l = -j-1 \mod k$ we say that l and j are in *symmetric residue classes* modulo k. The function ϕ_k maps integers that are in the same residue class or in symmetric residue classes modulo k to the same value. We say that such integers are in the same *extended residue class* modulo k (this is an equivalence relation on the integers).

At the beginning of round i the adversary will maintain an integer k_i which is a possible palindromic-period length. The adversary answers the comparisons of round i in such a way that some k_{i+1} is a possible palindromic-period length and few symbols of $\mathcal{S}$ are fixed. Let $K_i = n^{1-4^{-(i-1)}}$. The adversary will maintain the following invariants which hold at the beginning of round number i:

1. k_i satisfies $\frac{1}{2}K_i \le k_i \le K_i$.
2. If $\mathcal{S}[l]$ was fixed then for every j such that $\phi_{k_i}(l) = \phi_{k_i}(j)$, $\mathcal{S}[j]$ was fixed to the same symbol. In other words, the entire extended residue class of l modulo k_i was fixed to the same symbol.
3. If a comparison was answered as equal then both symbols compared were fixed to the same value.
4. If a comparison between positions l and j was answered as unequal, then
   a. l and j are in different extended residue classes modulo k_i. That is $\phi_{k_i}(l) \ne \phi_{k_i}(j)$.
   b. if the symbols were fixed then they were fixed to different values.
5. The number of fixed symbols f_i satisfies $f_i \le K_i$.

Note that invariants 3 and 4 imply consistency of the answers given so far. Invariants 2, 3 and 4 imply that k_i is a possible palindromic-period length: if we fix all symbols in each unfixed extended residue class modulo k_i to a new symbol, using the same symbol within an extended residue class but different symbols for unrelated residue classes, we obtain a string consistent with the answers given so far that has a palindromic-period length k_i. Such a string will have initial palindromes of all lengths which are integral multiples of k_i.

We start at round number 1 with $k_1 = K_1 = 1$. It is easy to see that the invariants hold initially. We show how to answer the comparisons of round i and how to choose k_{i+1} so that the invariants still hold. All multiples of k_i in the range $\frac{1}{2}K_{i+1} \ldots K_{i+1}$ are candidates for the new k_{i+1}. A comparison $\mathcal{S}[l] = \mathcal{S}[j]$ must be answered as equal if l and j are in the same extended residue class modulo k_{i+1}, that is, if $\phi_{k_{i+1}}(l) = \phi_{k_{i+1}}(j)$. We say that k_{i+1} *forces* this comparison.

Lemma 5.2 *If $p, q, r \geq \sqrt{\frac{2n}{k_i}}$ and are relatively prime, then a comparison $\mathcal{S}[s] = \mathcal{S}[t]$ is forced by at most two of pk_i and qk_i and rk_i.*

Proof: A comparison can be forced by some pk_i because indices of the compared symbols are in the same residue class or because there are in symmetric residue classes.

Assume s and t are in the same residue classes modulo pk_i and qk_i, thus $s \equiv t \mod pk_i$ and $s \equiv t \mod qk_i$. Then also $s \equiv t \mod pqk_i$. But $pqk_i \geq n$ and $0 \leq s, t \leq n$ what implies that $s = t$, a contradiction.

If s and t are in symmetric residue classes modulo pk_i and qk_i then $s \equiv -t-1 \mod pk_i$ and $s \equiv -t-1 \mod qk_i$. Then also $s+t+1 \equiv 0 \mod pqk_i$. But $pqk_i > 2n$ and $0 \leq s, t \leq n$, a contradiction.

The only remaining case is when s and t are in the same residue class modulo one of pk_i and qk_i and in symmetric residue classes modulo the other. In this case we look at the third candidate rk_i and consider the pairs rk_i and pk_i or rk_i and qk_i. One of these pairs is in one of the categories above, a contradiction to the existence of the third candidate. □

Lemma 5.3 *The number of candidates for k_{i+1} which are prime multiples of k_i and satisfy $\frac{1}{2}K_{i+1} \leq k_{i+1} \leq K_{i+1}$ is greater than $\frac{K_{i+1}}{4K_i \log n}$. Each such candidate satisfies the condition of Lemma 5.2.*

Proof: These candidates are of the form pk_i for prime p. The number of such prime values of p can be estimated using Corollary 3.12 It is at least

$$\frac{1}{4}\frac{K_{i+1}}{k_i \log \frac{K_{i+1}}{k_i}} \geq \frac{K_{i+1}}{4K_i \log n}.$$

Each one of these candidates also satisfies the condition of Lemma 5.2 since $k_i \leq K_i$, $pk_i \geq \frac{K_{i+1}}{2}$ and

$$p^2 \geq \frac{1}{k_i}\frac{K_{i+1}^2}{4K_i} = \frac{1}{k_i}\frac{n^{2-2\cdot 4^{-i}}}{4n^{1-4^{-(i-1)}}} = \frac{n}{k_i}\frac{1}{4}n^{2\cdot 4^{-i}} \geq \frac{2n}{k_i}.$$

□

Lemma 5.4 *There exists a candidate for* k_{i+1} *in the range* $\frac{1}{2}K_{i+1} \dots K_{i+1}$ *that forces at most* $\frac{8nK_i \log n}{K_{i+1}}$ *comparisons.*

Proof: By Lemma 5.3 there are at least $\frac{K_{i+1}}{4K_i \log n}$ such candidates which are prime multiples of k_i and satisfy the condition of Lemma 5.2. By Lemma 5.2 each of the n comparisons is forced by at most two of them. So the total number of comparisons forced by all these candidates is at most $2n$ (at most 2 comparisons forced by each candidate). Thus, there is a candidate that forces at most $\frac{8nK_i \log n}{K_{i+1}}$ comparisons. □

Lemma 5.5 *For* n *large enough and* $i \leq \frac{1}{4}\log\log n$, $1 + n^{2\cdot 4^{-i}} 64 \log n \leq n^{3\cdot 4^{-i}}$.

Proof: For n large enough,

$$\log\log(1 + 64\log n) < \frac{1}{2}\log\log n = (1 - \frac{2}{4})\log\log n$$

$$\log(1 + 64\log n) < 4^{-\frac{1}{4}\log\log n}\log n$$

$$1 + 64\log n < n^{4^{-\frac{1}{4}\log\log n}} \leq n^{4^{-i}},$$

from which the lemma follows. □

Lemma 5.6 *Assume the invariants hold at the beginning of round* i *and the adversary chooses* k_{i+1} *to be a candidate which forces at most* $\frac{8nK_i \log n}{K_{i+1}}$ *comparisons. Then the adversary can answer the comparisons in round* i *so that the invariants also hold at the beginning of round* $i + 1$.

Proof: By Lemma 5.4 such k_{i+1} exists. For each comparison that is forced by k_{i+1} and is of the form $\mathcal{S}[l] = \mathcal{S}[j]$ where $\phi_{k_{i+1}}(l) = \phi_{k_{i+1}}(j)$ the adversary fixes the residue class modulo k_{i+1} and its symmetric residue class (the extended residue class) to the same new symbol (a different symbol for different extended residue classes). The adversary answers comparisons between fixed symbols based on the value they are fixed to. All other comparisons involve symbols that are not in the same extended residue class modulo k_{i+1} (and at least one unfixed symbol) and are always answered as unequal.

The extended residue classes form a partition of the set of integers between 0 and n. This partition is refined when we move from extended residue classes modulo k_i to extended residue classes modulo k_{i+1}. Since k_{i+1} is a multiple of k_i, the extended residue classes modulo k_i split. This means that if two indices are in different extended residue classes modulo k_i, then they are also in different extended residue classes modulo k_{i+1}; and if two indices are in the same extended residue class modulo k_{i+1}, then they are also in the same extended residue class modulo k_i.

We show that the invariants still hold.

1. The candidate we chose for k_{i+1} was in the required range.

2. Extended residue classes which were fixed before split into several extended residue classes, all are fixed. Any symbol fixed at this round causes its entire extended residue class modulo k_{i+1} to be fixed to the same symbol.

3. Equal answers of previous rounds are not affected since the symbols involved were fixed to the same value by the invariants held before. Equal answers of this round are either between symbols which were fixed before to the same value or are within the same extended residue class modulo k_{i+1} and the entire extended residue class is fixed to the same value.

4. **a.** Unequal answers that were given in previous rounds are between different extended residue classes modulo k_{i+1} since extended residue classes modulo k_i split. Unequal answers of this round are between different extended residue classes because comparisons within the same extended residue class modulo k_{i+1} are always answered as equal.

   b. Unequal answers which involve symbols which were fixed before this round are consistent because fixed values dictate the answers to the comparisons. Unequal answers which involve symbols that are fixed at the end of this round and at least one was fixed at this round are consistent since a new symbol is used for each extended residue classes fixed.

5. We prove inductively that $f_{i+1} \leq K_{i+1}$. We fix at most $\frac{16nK_i \log n}{K_{i+1}}$ residue classes modulo k_{i+1}. There are k_{i+1} such classes and each class has at most $\lceil \frac{n}{k_{i+1}} \rceil \leq \frac{2n}{k_{i+1}}$ elements. By Lemma 5.5 and simple algebra the number of fixed elements satisfies:

$$\begin{aligned}
f_{i+1} &\leq f_i + \frac{2n}{k_{i+1}} \frac{16nK_i \log n}{K_{i+1}} \\
&\leq K_i \left[1 + \left(\frac{n}{K_{i+1}} \right)^2 64 \log n \right] \\
&\leq n^{1-4^{-(i-1)}} (1 + n^{2 \cdot 4^{-i}} 64 \log n) \\
&\leq n^{1-4^{-i}} = K_{i+1}.
\end{aligned}$$

□

Theorem 5.7 *Any comparison-based parallel algorithm for finding the initial palindromes of a string $\mathcal{S}[0..n]$ using n comparisons in each round requires $\frac{1}{4} \log \log n$ rounds. If p comparisons are allowed in each round, then $\Omega(\log \log_{\frac{2p}{n}} p)$ rounds are required.*

Proof: Fix an algorithm which finds the initial palindromes of $\mathcal{S}$ and let the adversary described above answer the comparisons. After $i = \frac{1}{4} \log \log n$ rounds $f_{i+1}, k_{i+1} \leq$

$n^{1-4^{-\frac{1}{4}\log\log n}} = \frac{n}{2^{\sqrt{\log n}}} \leq \frac{n}{2}$. The adversary can still fix $\mathcal{S}$ to have a palindromic-period length k_{i+1} by fixing each remaining residue class modulo k_{i+1} and its symmetric residue class to the same symbol, different symbol for each class. In this case any integral multiple of k_{i+1} is also an initial palindrome. Alternatively, the adversary can fix all unfixed symbols to different symbols. Note that this choice is consistent with all the the comparisons answered so far by invariants 3 and 4, and the string does not have any initial palindrome of length larger than $\frac{n}{2}$. In fact, in the latter case, the string will not have any initial palindrome except the trivial of length one. Consequently, any algorithm which terminates in less than $\frac{1}{4}\log\log n$ rounds can be fooled. This proof actually gives a lower bound also for computing the period length.

It remains to show that if the p comparisons are allowed at each round, then the any algorithm takes $\Omega(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ rounds. If $p < \frac{n}{\log\log n}$, then $\Omega(\frac{n}{p})$ rounds are required. If $\frac{n}{\log\log n} \leq p \leq n$ then the bound above is still applies. If $p > n^2$, then the simple algorithm of Lemma 3.2 solves the problem in constant time. The only remaining case is when $n \leq p \leq n^2$.

We change n to p in the appropriate places of the lower bound proof above. In particular we choose $K_i = p^{1-4^{-(i-1)}}$. The adversary can go on as long as $K_i \leq \frac{n}{2}$. By Lemma 3.7, $\Omega(\log\log_{\frac{2p}{n}} p)$ rounds are necessary.

By combining all these bounds into a single expression we get the promised bound of $\Omega(\lceil\frac{n}{p}\rceil + \log\log_{\lceil 1+p/n\rceil} 2p)$ rounds. $\Box$

Theorem 5.8 *The lower bound holds also for order comparisons.*

Proof: The adversary gradually defines the linear order of the symbols. He does it in such a way that the answers to comparisons in round i are determined at the round or before. The order is determined by a lexicographic order on a name given to each symbol and is extended for unfixed symbols at each round.

At round i, after k_{i+1} is chosen and before fixing new symbols, all names of unfixed symbols $\mathcal{S}[l]$ are extended to be $<\phi_{k_2}(l), \phi_{k_3}(l), ..., \phi_{k_{i+1}}(l)>$. If a symbol $\mathcal{S}[l]$ was fixed at round $j \leq i$ its entire extended residue class and its symmetric residue class are assigned the the name $<\phi_{k_2}(l), \phi_{k_3}(l), ..., \phi_{k_{j+1}}(l)>$ which is never changed. Note that all other names can not have this name as a prefix. If two symbols compared at round i are in the same or symmetric residue classes modulo k_{i+1}, then they were fixed at this round or before and given the same name. Symbols which are compared at round i and are in different extended residue classes modulo k_{i+1} have different name unless they were fixed earlier to the same value. The result of the comparison ($<, =, >$) is already determined even if the full final name of the symbols is not defined yet. At the end, if the extended residue classes modulo k_{i+1} are fixed in such a way that k_{i+1} is the palindromic-period length, then the full names are given as before. If all symbols are fixed to be different, the name given to unfixed position l is $<\phi_{k_2}(l), \phi_{k_3}(l), ..., \phi_{k_{i+1}}(l), l>$. $\Box$

5.4 Open Problems

There are still some open problems involving palindromes in parallel computation.

1. Does a fast parallel algorithm that finds all palindromes of a string exist? The best result so far is an $O(\log n)$ time n-processor algorithm of Crochemore and Rytter [40] .

2. Does a fast parallel algorithm that recognizes strings that are in Palstar exist? The best result so far is an $O(\log n)$ time n-processor algorithm [40] that works only for some special cases.

Chapter 6

Conclusion

The research reported in this thesis has been focused on theoretical aspects of algorithm involving strings.

In sequential computation we studied an interesting combinatorial problem of the number of comparisons required for string matching. Apart from the theoretical interest in this problem, we believe that determining the exact complexity of string matching and gaining deeper understanding of the structure of the problem may lead to better practical algorithms. Although linear-time string matching algorithms exist for quite some time, there is a hugh difference between the efficiency of the known algorithms [58].

In parallel computation we studied the complexity of the string matching problem and three other problems and provided tight lower and upper bounds under the general alphabet assumption. We were able to obtain optimal algorithms for the three other string problems by using the parallel string matching algorithm we developed. The lower bounds for these problems were also obtained by modifications to the lower bound for string matching.

We suspect that the technique of solving several string matching problems in parallel and combining the results to get answers to other string problem can be used to solve many other string problems.

Bibliography

[1] A. V. Aho. Algorithms for finding pattern in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 257–300. Elsevier Science Publishers B. V., Amsterdam, the Netherlands, 1990.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, 1975.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA., 1974.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA., 1983.

[5] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, NATO ASI Series F, Vol 12, pages 85–96. Springer-Verlag, Berlin, Germany, 1984.

[6] A. Apostolico. Efficient CRCW-PRAM Algorithms for Universal Substring Searching. Technical Report CS-TR-91–009, Dept. of Computer Science, Purdue University, 1991.

[7] A. Apostolico. Optimal parallel detection of squares in strings. Technical Report CS-TR-91–026, Dept. of Computer Science, Purdue University, 1991.

[8] A. Apostolico and D. Breslauer. On the exact complexity of comparison based string matching. manuscript, 1992.

[9] A. Apostolico, D. Breslauer, and Z. Galil. Optimal Parallel Algorithms for Periods, Palindromes and Squares. In *Proc. 19th International Colluquium on Automata, Languages, and Programming*. Springer-Verlag, Berlin, Germany, 1992. To appear.

[10] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

[11] A. Apostolico, C. Ilioupulis, G. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree, with applications. *Algorithmica*, 3:98–105, 1988.

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

[13] 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.

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

[15] S. W. Bent and J. W. John. Finding the median requires 2n comparisons. In *Proc. 17th ACM Symp. on Theory of Computing*, pages 213–216, 1985.

[16] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizeable problems. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 309–319, 1989.

[17] O. Berkman, B. Schieber, and U. Vishkin. Some doubly logarithmic optimal parallel algorithms based on finding nearest smallers. manuscript, 1988.

[18] O. Berkman and U. Vishkin. Recursive $*$-tree parallel data-structure. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 196–202, 1989.

[19] R. Boppana. Optimal separation between concurrent-write parallel machines. In *Proc. 21st ACM Symp. on Theory of Computing*, pages 320–326, 1989.

[20] A. Borodin and J. E. Hopcroft. Routing, merging and sorting on parallel models of comparison. *J. Comput. System Sci.*, 30:130–145, 1985.

[21] A. B. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa. A time-space tradeoff for sorting on non-oblivious machines. In *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 294–301, 1979.

[22] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20:762–772, 1977.

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

[24] D. Breslauer. An optimal $O(\log\log n)$ time parallel string matching algorithm. Master's thesis, Dept. of Computer Science, Tel-Aviv University, Tel-Aviv, Israel, 1989.

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

[26] D. Breslauer and Z. Galil. Closing the Gap in the Exact Complexity of String Matching. manuscript, 1992.

[27] D. Breslauer and Z. Galil. Finding all periods and initial palindromes of a string in parallel. manuscript, 1992.

[28] Y. Cesari and M. Vincent. Une caractérisation des mots periodiques. *C.R. Acad. Sci. Paris*, 286(A):1175–1177, 1978.

[29] R. Cole. Tight bounds on the complexity of the Boyer-Moore pattern matching algorithm. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 224–233, 1991.

[30] R. Cole and U. Vishkin. Faster optimal prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[31] L. Colussi. Combinatorial properties of strings. manuscript, 1991.

[32] L. Colussi. Correctness and efficiency of string matching algorithms. *Inform. and Control*, 95:225–251, 1991.

[33] S. A. Cook. Linear time simulation of deterministic two-way pushdown automata. In *Information Processing 71*, pages 75–80. North Holland Publishing Co., Amsterdam, the Netherlands, 1972.

[34] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM J. Comput.*, 15(1):87–97, 1986.

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

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

[37] M. Crochemore. String-matching and periods. *Bulletin of EATCS*, October 1989.

[38] M. Crochemore and D. Perrin. Two-way string-matching. *J. Assoc. Comput. Mach.*, 38(3):651–675, 1991.

[39] M. Crochemore and W. Rytter. Efficient parallel algorithms to test square-freeness and factorize strings. *Inform. Process. Lett.*, 28:57–60, 1991.

[40] 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.

[41] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. In *Ann. Rev. Comput. Sci.*, volume 3, pages 233–283, 1988.

[42] 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.

[43] 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.

[44] L. R. Ford and S. M. Johnson. A tournament problem. *American Mathematical Monthly*, 66:387–389, 1959.

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

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

[47] Z. Galil. A Constant-Time Optimal Parallel String-Matching Algorithm. In *Proc. 24th ACM Symp. on Theory of Computing*, page to appear, 1992.

[48] Z. Galil and R. Giancarlo. The exact complexity of string matching: upper bounds. manuscript, 1991.

[49] Z. Galil and R. Giancarlo. On the exact complexity of string matching: lower bounds. *SIAM J. Comput.*, 20(6):1008–1020, 1991.

[50] Z. Galil and J. Seiferas. A Linear-Time On-Line Recognition Algorithm for "Palstar". *J. Assoc. Comput. Mach.*, 25(1):102–111, 1978.

[51] Z. Galil and J. Seiferas. Saving space in fast string-matching. *SIAM J. Comput.*, 2:417–438, 1980.

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

[53] M. Geréb-Graus and M. Li. Three one-way heads cannot do string matching. manuscript, 1990.

[54] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, Cambridge, Great Britain, 1988.

[55] L. Guibas and A. M. Odlyzko. Periods in strings. *Journal of Combinatorial Theory, Series A*, 30:19–42, 1981.

[56] T. Hagerup. Constant-Time Parallel Integer Sorting. In *Proc. 23rd ACM Symp. on Theory of Computing*, pages 299–306, 1991.

[57] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA., 1979.

[58] A. Hume. A tale of two greps. *Software Practice and Experience*, 18:1063–1072, 1988.

[59] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. 4th ACM Symp. on Theory of Computing*, pages 125–136, 1972.

[60] R. M. Karp and M. O. Rabin. Efficient randomized pattern matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, 1987.

[61] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 869–941. Elsevier Science Publishers B. V., Amsterdam, the Netherlands, 1990.

[62] Z. Kedem, G. M. Landau, and K. Palem. Optimal parallel suffix-prefix matching algorithm and applications. manuscript, 1988.

[63] D. G. Kirkpatrick. Topics in the complexity of combinatorial algorithms. Technical report, Dept. of Computer Science, University of Toronto, Toronto, Canada, 1974.

[64] D. G. Kirkpatrick. A unified lower bound for selection and set partitioning problems. *J. Assoc. Comput. Mach.*, 28:150–165, 1981.

[65] S. S. Kislitsyn. On the selection of the kth element of an ordered set by pairwise comparison. *Sibirskii Mat. Zhurnal*, 5:557–564, 1964.

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

[67] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE trans. on computers 32*, 32:942–946, 1983.

[68] M. Li. Lower bounds on string-matching. Technical Report TR 84–63, Cornell University, Department of Computer Science, 1984.

[69] M. Li and Y. Yesha. String-matching cannot be done by a two-head one-way deterministic finite automaton. *Inform. Process. Lett.*, 22:231–235, 1986.

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

[71] 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.

[72] 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.

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

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

[75] E. M. McCreight. A space economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.*, 23:262–272, 1976.

[76] I. Pohl. A sorting problem and its complexity. *Comm. of the ACM*, 15:462–464, 1972.

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

[78] P. Ragde. The parallel simplicity of compaction and chaining. In *Proc. 17th International Colluquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 774–751. Springer-Verlag, Berlin, Germany, 1990.

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

[80] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[81] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *J. Comput. System Sci.*, 13:184–199, 1976.

[82] J. Schreier. On tournament elimination systems. *Mathesis Polska*, 7:154–160, 1932.

[83] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *J. Algorithms*, 2:88–102, 1981.

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

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

[86] L. G. Valiant. Parallelism in comparison models. *SIAM J. Comput.*, 4:348–355, 1975.

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

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

[89] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[90] C. K. Yap. New upper bounds for selection. *Comm. of the ACM*, 19:501–508, 1979.

[91] U. Zwick and M. S. Paterson. Lower bounds for string matching in the sequential comparison model. manuscript, 1991.