東北大学機関リポジトリ
TOUR
Tohoku University Repository

Duel-and-Sweep Algorithms for String Pattern Matching Problems under Substring Consistent Equivalence Relations

著者	Jargalsaikhan Davaajav
学位授与機関	Tohoku University
学位授与番号	11301甲第20473号
URL	http://hdl.handle.net/10097/00135842

Duel-and-sweep algorithms for string pattern matching problems under substring consistent equivalence relations

Davaajav Jargalsaikhan

Graduate School of Information Sciences
Tohoku University, Japan

This dissertation is submitted in partial fulfillment of the requirements for the degree of *Doctor of Philosophy*

16 February 2022

Acknowledgment

First of all, I would like to express my profound gratitude to Prof. Ayumi Shinohara, Prof. Ryo Yoshinaka and Prof. Diptarama Hendrian, my research supervisors, for their patience and constant support in overcoming numerous obstacles I have been facing during the 9 years of my study and research at Tohoku University.

I would also like to express my appreciation to my dissertation committee: Prof. Xiao Zhou and Prof. Masanori Hariyama, for their insightful suggestions, feedbacks, and comments which helped me to write a better dissertation.

My deepest appreciation also goes to staffs and colleagues of Shinohara-Yoshinaka Laboratory for their insightful feedback and encouragement and Naomi Ogasawara who provided a lot of helpful supports on organizing documents and research fund.

I would also like to express my gratitude to Tohoku University Division for Interdisciplinary Advanced Research and Education, for their support in scholarship and research grant during my Ph.D. study.

Lastly, I would like to thank my family for supporting me throughout my years of study and through the process of research and writing this dissertation. This accomplishment would not have been possible without them.

Davaajav Jargalsaikhan

16 February 2022

Abstract

In this dissertation, we propose efficient algorithms based on the dueling technique for various pattern matching problems. We propose serial and parallel duel-and-sweep algorithms that are generalized for substring consistent equivalence relations (SCER). SCER is a generalization of different matchings, that include exact, parameterized, order-preserving, palindrome and cartesian-tree matching. Also, we propose instance-specific algorithms that solve the pattern searching problem for the above matchings.

For SCER, we have proposed new serial and parallel algorithms for SCER pattern matching problems. Given a text of length n and a pattern of length m, we assume that encodings of the text and the pattern can be encoded in $O(\tau_n + \tau_m)$ time in serial, respectively. Assuming that the encoding has been computed, we assume that re-encoding the element at position k can be re-encoded, with respect to a substring of length n, using ξ_m time. Our serial algorithm for one-dimensional SCER runs in $O(n \cdot \xi_m + m \log m \cdot \xi_m)$ time. We also give serial algorithms for parameterized, cartesian-tree, palindrome and order-preserving matchings that run in $O(n)$ time. For cartesian-tree and palindrome matchings, we give a preprocessing algorithm that runs in $O(m)$ time. For parameterized matching the preprocessing runs in $O(m \log |\Pi|)$ time. Here, $|\Pi|$ is the size of the variable alphabet. For order-preserving matching, the preprocessing runs in $O(m \log m)$ time. Also, for order-preserving matching, assuming that a text is a two-dimensional string of size $n \times n$ and a pattern is a two-dimensional string of size $m \times m$, our serial algorithm that solves the 2d matching problem in $O(n^2 m + m^3)$ time.

For our parallel algorithms we use Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM). Our parallel algorithms are the first algorithms to solve various SCER pattern matching problems in parallel. Our parallel algorithm for SCER run in $O(\log^3 m \cdot \xi^{\mathrm{t}}_m)$ time using $O(n \log^2 m \cdot \xi^{\mathrm{w}}_m)$ work on the P-CRCW PRAM, with $O(\log^2 m \cdot \xi^{\mathrm{t}}_m)$ time and $O(m \log^2 m \cdot \xi^{\mathrm{w}}_m)$ work preprocessing. For parameterized, palindrome and order-preserving matchings, we give parallel algorithms that run in $O(\log^3 m)$ time using $O(n \log^2 m)$ work on the P-CRCW PRAM, with $O(\log m)$ time and $O(m \log m)$ work preprocessing. For cartesian-tree matching, the preprocessing takes $O(\log^2 m)$ time and $O(m \log^2 m)$ work. We also describe how to compute encodings for these matching for order-preserving matching in $O(\log n)$ time and $O(n \log n)$ work in parallel, where n is the length of the string to be encoded.

Contents

1 Introduction **1**

1.1 Background . . . 1

1.2 Contributions . . . 7

2 Preliminaries **11**

2.1 Notations . . . 11

2.2 Substring consistent equivalence relations . . . 12

2.3 The model of parallel computation . . . 15

3 Serial duel-and-sweep algorithms for SCER **18**

3.1 Pattern matching for SCER . . . 18

3.2 Overview of duel-and-sweep algorithm . . . 18

3.2.1 Pattern preprocessing . . . 21

3.2.2 Pattern searching . . . 23

3.3 Parameterized matching . . . 25

3.4 Cartesian-tree matching . . . 27

3.5 Palindrome matching . . . 28

3.6 Order-preserving matching . . . 32

3.6.1 One-dimensional matching . . . 32

3.6.2 Two-dimensional matching . . . 39

4 Parallel duel-and-sweep algorithms for SCER **45**

4.1 Aperiodic pattern case for SCER 46

4.2 General case for SCER 56

4.2.1 Pattern preprocessing 56

4.2.2 Pattern searching 63

4.3 Reversible SCER 75

4.4 Parameterized matching 79

4.5 Cartesian-tree matching 80

4.6 Palindrome matching 81

4.7 Order-preserving matching 84

5 Conclusion and Future Work **88**

References **95**

List of Publications **96**

Chapter 1

Introduction

1.1 Background

The string matching problem is a fundamental and a widely studied problem in computer science. Given a text and a pattern, the string matching problem searches for all substrings of the text that match the pattern. String matching algorithms find practical use for solving problems in information retrieval, such as text mining, natural language processing, image processing, speech processing. Recent trend of information retrieval focuses on detecting and retrieving text in images, video, documents and social media. Natural language processing is an integral part of multimedia information retrieval. After retrieval methods recognize text using an optical character recognizer, they use string matching algorithms to search for relevant words in the database. For digitized texts, such as annotated data of images or videos, the methods use string matching to define context for extracting relevant words at a high level from multimedia databases. There are methods in [24], where string matching is used to index and retrieve information from multimedia databases at a high level.

Bioinformatics is another field where the string matching algorithms are widely used. Molecular biology information, for the purpose of this paper, consists of sequences of DNA/RNA or sequences of amino acids called proteins. DNA/RNA sequences can be

regarded as strings consisting of 4 different symbols, and protein sequences can be regarded as strings consisting of 20 different symbols. String matching algorithms are used for finding query sequences (the pattern) from the large sequence datasets (the text). One of the central problems in bioinformatics that continues to draw attention from researchers for the past few decades is the protein structure prediction problem. That is, given an amino-acid sequence of a protein, our task is to predict its spatial structure. Some methods that are proven to be comparatively successful in solving the protein structure prediction problem extensively use string matching algorithms. First, the spatial information for short protein sequences is experimentally established and stored in a database. Generally, proteins that have similar amino-acid sequence tend to have similar spatial structure. Given a new amino-acid sequence, which is much longer than those in the dataset, the algorithm finds fragments in the input sequence that match with some sequence in the dataset. Finally, the algorithm predicts the overall structure of the protein by combining the spatial information of the fragments.

Many exact pattern matching algorithms have been proposed such as the well-known Knuth-Morris-Pratt algorithm [36], Boyer-Moore algorithm [9], and Horspool algorithm [31]. These algorithms preprocess the pattern first and then match the pattern from its prefix or suffix when comparing it with the text. Vishkin proposed two algorithms for pattern matching, pattern matching by duel-and-sweep [44] and pattern matching by sampling [45]. Both algorithms match the pattern to a substring of the text from some positions which are determined by the property of the pattern, instead of its prefix or suffix. These algorithms are developed also for parallel processing.

Over the years, the classic exact pattern matching problem has been modified and extended to meet various needs that arise from different real world problems. One natural extension of the exact pattern matching is two-dimensional pattern matching. Another direction for extensions is to modify the matching function. Perhaps, the most natural modification of the matching function is approximate matching. Given a text and a

pattern, the task of the approximate matching is to find all the occurrences of pattern in the text whose edit distance to the pattern is at most k. The edit distance between two strings is defined as minimum number of character insertion, deletion and replacements needed to make them equal.

In parameterized matching, introduced by Baker [5, 6], two strings of equal length match, if there exists an one-one mapping of their characters. Two strings X and Y of equal length parameterize match, or p-match for short, if there is a bijection π from the alphabet of X to the alphabet of Y such that $Y[i] = \pi(X[i])$ for every $1 \leq i \leq |X|$. In the parameterized matching problem, given an input composing a text T and a pattern P, the goal is to find all the substrings of T of length $|P|$ that p-match P. Parameterized matching was introduced for applications that arise in software tools for analyzing source code. Specifically, the application is to identify duplicate code in large software systems for reuse. Here it is desirable to find not only exact matches between program fragments but also parameterized matches, namely, where the two program fragments are equal but possibly use interchangeable identifiers (representing variable, constant, or function names). Amir et al. [2] presented tight bounds for parameterized matching in the presence of an unbounded size alphabet.

Order-preserving matching [35, 37] considers the relative order of elements, rather than their real values. In order-preserving matching, two strings of equal length match, if the relative of the elements are the same. For instance, for exact matching $(12, 35, 5) \neq (25, 30, 21)$. However, for OPPM, $(12, 35, 5)$ matches $(25, 30, 21)$, since the relative order of the elements is same. Namely, the first element is the second smallest, the second element is the largest and the third element is the smallest among $(12, 35, 5)$, $(25, 30, 21)$, respectively. One of motivations is given by the following scenario. Consider a sequence of numbers that models a time series which is known to repeat the same shape every fixed period of time. For example, this could be certain stock market data or statistics data from a social network that is strongly dependent on the day of the week, i.e., repeats the

same shape every consecutive week. Order-preserving matching has gained much interest in recent years, due to its applicability in problems where the relative order matters, such as share prices in stock markets, weather data or musical notes.

Kubica et al. [37] and Kim et al. [35] proposed the solution for the order-preserving pattern matching problem based on the KMP algorithm. Their KMP-based algorithm runs in $O(n + m \log m)$ time. Cho et al. [16] brought forward another algorithm based on the Horspool algorithm that uses q-grams, which was proven to be experimentally fast. Crochemore et al. [23] proposed useful data structures for OPPM. On the other hand, Chhabra and Tarhio [15], Faro and Külekci [25] proposed filtration methods which are practically fast. Moreover, faster filtration algorithms using SIMD (Single Instruction Multiple Data) instructions were proposed by Cantone et al. [13], Chhabra et al. [14] and Ueki et al. [43]. They showed that SIMD instructions are effective in speeding up their algorithms.

An equivalence relation that is closely related to the order-preserving matching is cartesian-tree matching [41]. Cartesian-tree matching also concerns the relative order of elements within a string. The difference between order-preserving matching and cartesian-tree matching is that order-preserving matching considers global relative order, while cartesian-tree matching considers local relative order. For instance, $(10, 5, 7)$ and $(13, 10, 17)$ does not match for order-preserving matching, but they match for cartesian-tree matching. Park et al. [41] proposed a linear time algorithm to solve cartesian-tree matching.

Another interesting matching that is worth considering is palindrome matching [42]. Palindrome is a symmetric string that reads same from left-to-right and right-to-left. Two strings of same length are said to be pal-equivalent iff the length of the maximal palindrome at every center in the strings is equal. Palindrome matching is useful for bioinformatics when finding patterns in DNA or RNA sequences.

The difficulty of non-classical matchings mainly comes from the fact that we cannot determine the isomorphism by comparing the symbols in the text and the pattern on each

position independently; instead, we have to consider their respective relative orders in the pattern and in the text. For instance, consider strings X_1, X_2, Y_1, Y_2 of equal length. Suppose that X_1 matches Y_1 and X_2 matches Y_2. In exact matching, the concatenation of X_1 and X_2 will match that of Y_1 and Y_2. However, in parameterized matching or order-preserving matching, the two concatenations will not necessarily match each other.

Many matching functions that are used in different string matching problems, including exact, parameterized, order-preserving, cartesian-tree and palindrome matchings, fall under the class of subsequence consistent equivalence relations (SCER). An equivalence relation on strings is called a substring consistent equivalence relation, if for any two strings X, Y of equal length, the equivalence relation satisfies the following: X matching Y under SCER implies that any substrings of X and Y match. That is, if we denote the substring of X and Y that starts at position i and ends at position j, as $X[i : j]$ and $Y[i : j]$, respectively, then $X[i : j]$ matches $Y[i : j]$, for all $1 \leq i \leq j < |X|$. SCER was introduced by [39], and it is the first attempt to analyze different pattern matching problems under the same general theory.

The first efficient parallel string matching was by Galil [27], where a framework benefiting from periodicity properties in strings was introduced. Similar properties were used in later parallel string matching algorithms. The algorithm in Galil's original paper runs in logarithmic time and is optimal for an alphabet whose size is fixed. Vishkin [44] proposed a new idea that led to an optimal speed up algorithm regardless of the alphabet size. Breslauer and Galil [11] added that the dueling technique implies that the string matching problem is not more difficult, from the parallel algorithmic point of view, than the problem of finding the maximum among n elements. This enabled a $O(\log \log n)$ time parallel algorithm for the problem. In [45], the dueling idea was further extended the idea of deterministic sampling. In the dueling technique a single location is used to disqualify either of two candidate positions in the text. The deterministic sampling idea implies that there is a way for drastically disqualifying at once several candidate positions. As

the name suggests, the algorithm deterministically samples $\log m$ locations, where m is the length of the pattern, to disqualify candidate positions.

The dueling technique is proved to be useful not only in parallel pattern matching, but also in two-dimensional pattern matchings. The duel-and-sweep algorithm appeared in Amir et al. [2], where it was named “consistency and verification” and was used for two-dimensional exact matching; it is based on the dueling technique [44]. Cole et al. [19] extended it to two-dimensional parameterized matching.

It continues to be the case that there are only few parallel algorithms which are optimal, in spite of the interest in them. We list algorithms that introduced important techniques that are often used in parallel algorithms. See Cole and Vishkin [21] (among many others) for computation of prefix sums of n variables. In fact, it can be shown that replacing the summing operation with any other associative binary operation gives the same computational complexity. Cole and Vishkin [20] gives an optimal list ranking algorithm, that uses a technique called deterministic coin tossing. Deterministic coin tossing was used to eliminate randomness from the randomized list ranking algorithm. A tree contraction method that is used to efficiently evaluate expression trees in parallel was first proposed by [40]. The technique is easily generalized to arbitrary trees and used in graph-theoretic algorithms, such as maximum matching, minimum vertex cover and maximum independent set [30]. Borodin and Hopcroft [8] gave an estimation on the upper and lower bounds on the time of merging two sequences of length n using $2n$ processors, on the parallel comparison model. Cole [18] improves on the simple merging approach to sort array of n elements. In Cole's algorithm each node in the merging tree is associated with a list, which, at the end of the algorithm, should hold all elements, that it is responsible of, in the sorted order. His algorithm works on multiple levels of the tree at once, creating successively more refined approximations to the sorted lists that the nodes must eventually produce. For each node, the approximation to the final list can be obtained from the preceding approximation in constant time.

Table 1.1: Summary of proposed algorithms on each problems. The results on the parallel algorithms are obtained on P-CRCW PRAM. (For the parallel algorithms the value before comma indicates the time complexity, the value after the comma indicates the work complexity.)

<table>
<tr><th rowspan="2"></th><th colspan="2">Serial</th><th colspan="2">Parallel</th></tr>
<tr><th>Prep.</th><th>Search</th><th>Prep.</th><th>Search</th></tr>
<tr><td>SCER</td><td>$O(\tau_n + \xi_m m)$</td><td>$O(\xi_m n)$</td><td>$O(\tau^{\mathrm{t}}_n + \xi^{\mathrm{t}}_m \log^2 m)$,
$O(\tau^{\mathrm{w}}_n + \xi^{\mathrm{w}}_m m \log^2 m)$</td><td>$O(\xi^{\mathrm{t}}_m \log^3 m)$
$O(\xi^{\mathrm{w}}_m n \log^2 m)$</td></tr>
<tr><td>Cartes. tree</td><td rowspan="2">$O(m)$</td><td rowspan="4">$O(n)$</td><td>$O(\log^2 m)$,
$O(m \log^2 m)$</td><td rowspan="4">$O(\log^3 m)$,
$O(n \log^2 m)$</td></tr>
<tr><td>Palindrome</td><td rowspan="3">$O(\log m)$,
$O(m \log m)$</td></tr>
<tr><td>Order-preser.</td><td>$O(m \log m)$</td></tr>
<tr><td>Parameter.</td><td>$O(m \log \lvert\Pi\rvert)$</td></tr>
</table>

The family of models of computation used in this dissertation is the parallel random-access-machines (PRAMs). The PRAM model assumes that (1) the memory is uniformly shared among all processors; (2) there is no limit on the amount of shared memory; (3) issues such as synchronization and communication between processors are neglected. Our work focuses on the priority concurrent-read concurrent-write (P-CRCW) PRAM [32]. This model allows simultaneous reading from the same memory location as well as simultaneous writing. In case of multiple writes to the same memory cell, the P-CRCW PRAM grants access to the memory cell to the processor with the smallest index. When estimating the computational complexity of a parallel algorithm, it is custom to consider the parallel running time and the overall number of operations, which is referred to as work.

1.2 Contributions

In this dissertation, we propose efficient algorithms based on the dueling technique [44] for various pattern matching problems. We propose serial and parallel duel-and-sweep algorithms that are generalized for substring consistent equivalence relations (SCER) and consider specific instances of SCER that include parameterized matching, order-preserving

matching, palindrome matching and cartesian tree matching. For our parallel algorithms we use Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM) [32]. Our algorithm is theoretically as fast as the KMP-based algorithm for SCER [39]. Our parallel algorithms are the first algorithms to solve SCER in parallel. The details of our algorithms are as follows:

1. For SCER, we have proposed new serial and parallel algorithms for pattern matching problem. When considering SCER algorithms we make the following assumptions. For a string X, suppose that its encoding can be computed in $\tau_{|X|}$ time in serial and $\tau^{\mathrm{t}}_{|X|}$ time and $\tau^{\mathrm{w}}_{|X|}$ work in parallel on P-CRCW PRAM. Assuming that the encoding has been computed, we assume that re-encoding the element at position k with respect to suffix $X[i:]$, where $i \leq k$, takes $\xi_{|X[i:]|}$ time in serial and $\xi^{\mathrm{t}}_{|X|}$ time and $\xi^{\mathrm{w}}_{|X|}$ work in parallel on P-CRCW PRAM.

   Given a text of length n and a pattern of length m, for every algorithm in this paragraph, we assume that the encodings of the text and the pattern are computed using $O(\tau_n+\tau_m)$ time. Our serial algorithm for SCER runs in $O(n\cdot\xi_m+m\log m\cdot\xi_m)$ time.

   Our parallel algorithm run in $O(\log^3 m\cdot\xi^{\mathrm{t}}_m)$ time using $O(n\log^2 m\cdot\xi^{\mathrm{w}}_m)$ work on the P-CRCW PRAM, with $O(\log^2 m\cdot\xi^{\mathrm{t}}_m)$ time and $O(m\log^2 m\cdot\xi^{\mathrm{w}}_m)$ work preprocessing. If the pattern is aperiodic, we show that the pattern matching problem can be solved in $O(\log m\cdot\xi^{\mathrm{t}}_m)$ time using $O(n\cdot\xi^{\mathrm{w}}_m)$ work on the P-CRCW PRAM, with $O(\log m\cdot\xi^{\mathrm{t}}_m)$ time and $O(m\cdot\xi^{\mathrm{w}}_m)$ work preprocessing. Our parallel algorithm is the first algorithm that solves the pattern matching problem for SCER in parallel.

2. For parameterized matching, we have proposed efficient serial and parallel algorithms. For the parallel preprocessing, we have shown that it can be solved in $O(\log m)$ time and $O(m\log m)$ preprocessing time on P-CRCW PRAM. Also, we have proposed a parallel algorithm to compute the *prev*-encoding for parameterized

matching in $O(\log m)$ time using $O(m \log m)$ work on P-CRCW PRAM.

3. For cartesian-tree matching, we have proposed efficient serial and parallel algorithms. Our serial algorithm for parallel preprocessing, we have shown that it can be solved in $O(n)$ time. Our parallel algorithm runs in $O(\log^3 m)$ time and $O(n \log^2 m)$ work on P-CRCW PRAM.

4. For palindrome matching, we have proposed efficient serial preprocessing algorithm runs in $O(n)$ time. The serial pattern searching algorithm is also linear. Also, we have proposed a parallel algorithm runs in $O(\log^3 m)$ time and $O(n \log^2 m)$ work on P-CRCW PRAM with $O(\log m)$ time and $O(m \log m)$ work preprocessing.

5. For order-preserving matching, we have proposed serial and parallel algorithms that does not encode the text. Our serial algorithm for order-preserving matching solves the pattern matching problem in $O(n)$ time with $O(m \log m)$ time preprocessing. We show that our algorithm is theoretically as fast as the KMP algorithm for order-preserving pattern matching and faster than it in practice. Assuming that a text is a two-dimensional string of size $n \times n$ and a pattern is a two-dimensional string of size $m \times m$, our serial algorithm for two-dimensional order-preserving matching solves the 2d matching problem in $O(n^2 m + m^3)$ time. Our parallel algorithm for order-preserving matching solves the pattern matching problem in $O(\log^3 m)$ time using $O(n \log^2 m)$ work on P-CRCW PRAM with $O(\log m)$ time and $O(m \log m)$ preprocessing time on P-CRCW PRAM. Also, we have proposed a parallel computation of the nearest neighbor encoding for order-preserving matching that runs in $O(\log m)$ time using $O(m \log m)$ work on P-CRCW PRAM.

The rest of this dissertation is organized as follows. In Chapter 2, we describe the notations, the idea of duel-and-sweep algorithm and give definitions that we will use for SCER algorithms. Also, we describe the basics of the parallel computation models. In Chapter 3, we discuss our serial algorithms and consider serial algorithms for parameter-

ized, cartesian-tree,palindrome and order-preserving matchings. In Chapter 4, we propose parallel algorithms for one-dimensional SCER. Also, we describe parallel algorithms for computing the encodings and give optimizations for the pattern preprocessing for the matchings mentioned above. Lastly, we conclude our work in Chapter 5 and discuss future work that we might undertake.

Chapter 2

Preliminaries

2.1 Notations

We use Σ to denote an alphabet of integer symbols such that the comparison of any two symbols can be done in constant time. Σ^* denotes the set of strings over the alphabet Σ. For a string $X \in \Sigma^*$, the length of X is denoted by $|X|$. The *empty string*, denoted by ε, is a string of length 0. For a string $X \in \Sigma^*$ of length n, $X[i]$ denotes the i-th symbol of X, $X[i:j] = X[i]X[i+1]\ldots X[j]$ denotes a substring of X that begins at position i and ends at position j for $1 \leq i \leq j \leq n$. For convenience, we abbreviate $X[1:i]$ to $X[:i]$ and $X[i:n]$ to $X[i:]$, which are called *prefix* and *suffix* of X, respectively. Moreover, let $X[i:j] = \varepsilon$ if $i > j$. We denote the reverse of X as X^R.

Suppose that we are given a text T of length n and a pattern P of length m. For an integer x with $1 \leq x \leq n-m+1$, a *candidate* T_x is the substring of T starting from x of length m, i.e., $T_x = T[x:x+m-1]$. In the remainder of this paper, we fix text T to be of length n and pattern P to be of length m. We also assume that $n = 2m-1$. Larger texts can be cut into overlapping pieces of length $2m$ and processed independently. That is, for each $T[1:2m-1], T[m+1:3m-1], \ldots, T[n-2m+1:n]$, we process them independently.

2.2 Substring consistent equivalence relations

Over the past decades, different variations of the exact matching were studied, such as parameterized matching and order-preserving matching. Matsuoka et al. [39] generalized these matchings and defined a class of equivalence relations, called *substring consistent equivalence relations* (SCER).

Definition 2.1 (Substring consistent equivalence relation (SCER) [39])**.** *An equivalence relation* $\approx\ \subseteq \Sigma^* \times \Sigma^*$ *is a* substring consistent equivalence relation (SCER) *if for two string* X *and* Y*,* $X \approx Y$ *implies* $|X| = |Y|$ *and* $X[i:j] \approx Y[i:j]$ *for all* $1 \leq i \leq j \leq |X|$*.*

We say X $\approx$-matches Y iff $X \approx Y$. For instance, while the parameterized matching [5], order-preserving matching [37, 35] are SCERs, the permutation matching [12, 17] and function matching [1] are not. Matsuoka et al. [39] defined $\approx$-*occurrence* of a pattern P in a text T under an SCER $\approx$ as follows. Given a text T and a pattern P, a position i in T, $1 \leq i \leq n-m+1$, is an $\approx$-occurrence of P in T iff $P \approx T[i : i+m-1]$.

Definition 2.2 ($\approx$-pattern matching)**.**

Input: *A text* $T \in \Sigma^*$ *of length* n *and a pattern* $P \in \Sigma^*$ *of length* $m \leq n$*.*

Output: *All* $\approx$*-occurrences of* P *inside* T*.*

For non-classical matchings, such as parameterized matching and order-preserving matching, often it is convenient to encode the strings. These encodings are generalized for SCERs as follows.

Definition 2.3 ($\approx$-encoding)**.** *Let* Σ *and* Π *be alphabets. We say a function* $f : \Sigma^* \to \Pi^*$ *is an* $\approx$-encoding *if (1) for any string* $X \in \Sigma^*$, $|X| = |f(X)|$*, (2)* $f(X[1:i]) = f(X)[1:i]$*, (3) for two strings* X *and* Y *of equal length,* $f(X)[i] = f(Y)[i]$ *implies* $f(X[j+1:])[i-j] = f(Y[j+1:])[i-j]$ *for any* $j < i \leq |X| = |Y|$*, and (4)* $f(X) = f(Y)$ *iff* $X \approx Y$*.*

Examples of a $\approx$-encoding are the prev-encoding [5] for parameterized matching and parent-distance encoding [41] for cartesian-tree matching. The nearest neighbor encoding [35] for order-preserving matching is not a $\approx$-encoding, because the nearest neighbor encoding violates the third condition. However, by considering pairs of locations, instead of a single location, Jargalsaikhan et al. [33, 34] proposed duel-and-sweep algorithms that solve the order-preserving pattern matching problem using the nearest neighbor encoding. Matsuoka et al. [39] defined *$\approx$-prefix encoding*, which is $\approx$-encoding without Condition (3) of Definition 2.3. We will use $\approx$-prefix encoding for our serial SCER algorithm and use $\approx$-encoding for our parallel algorithm.

Amir and Kondratovsky [3] showed that there exists a $\approx$-encoding for any SCER [1]. By using a $\approx$-encoding, if $X[: i] \approx Y[: i]$ we can check whether $X[: i+1] \approx Y[: i+1]$ just by checking whether $f(X)[i+1] = f(Y)[i+1]$. For a string X and $\approx$-encoding f, let we denote $f(X)$ by $\widetilde{X}$ for simplicity. For convenience, we denote the encoding of element at position k with respect to $X[i :]$, as $\widetilde{X}_i[k]$. If $i = 1$, we omit 1 and $|X|$ in the notation and denote the encoding at position k as $\widetilde{X}[k]$.

Hereafter we fix an arbitrary SCER $\approx$. We say that a position i is a *tight mismatch position* if $X[1 : i-1] \approx Y[1 : i-1]$ and $X[1 : i] \not\approx Y[1 : i]$. For two strings X and Y, let $LCP(X, Y)$ be the length l of the longest prefixes of X and Y match. That is, l is the greatest integer such that $X[1 : l] \approx Y[1 : l]$. Obviously, if i is the tight mismatch position for $X \not\approx Y$, then $LCP(X, Y) = i - 1$. The converse holds if $i \leq \min\{|X|, |Y|\}$. Similarly, for a string X and an integer $0 \leq a < |X|$, we define $LCP_X(a) = LCP(X, X[a+1 : |X|])$. In other words, $LCP_X(a)$ is the length of the longest common prefix, when X is superimposed on itself with offset a.

Next, we review some periodicity properties for SCER. Matsuoka et al. [39] defined three types of periods: block-based, sliding-window-based and border-based periods. We will focus on the block-based and the border-based periods. Informally, an integer p is a

[1]Lemma 12 in [3] does not explicitly mention the third property, but their proof entails it.

block-based period of a string X if all blocks match each other, when X is partitioned into blocks of length p. An integer p is a border-based period of X, if the overlapping regions match each other, when X is copied and superimposed on itself with an offset p.

Definition 2.4 (Block-based period). *Given a string X of length n, positive integer p is called a* block-based period *of X, if*

$$X[1:p] \approx X[kp+1:kp+p] \text{ for } k \in \{1,\dots,\lfloor n/p \rfloor - 1\} \text{ and,}$$
$$X[1:r] \approx X[n-r+1:n] \text{ for } r = n \bmod p.$$

String X of length n is *block-periodic*, if there exists a block-based period $p \geq 2$ of X such that $n \geq 2p$. Otherwise, it is *block-aperiodic*.

Definition 2.5 (Border-based period). *Given a string X of length n, positive integer p is called a* border-based period *of X if $X[1:n-p] \approx X[p+1:n]$.*

For exact matching, if p is a border-based period of a string X, all block-based periods are also border-based periods. However, the reverse does not necessarily hold true for SCER [39]. For instance, in order-preserving matching $X = (13, 7, 10, 21, 14, 18, 20, 11, 15, 28, 22, 25)$ has a block-based period 3, since $X[1:3] \approx X[4:6] \approx X[7:9] \approx X[10:12]$. However, X does not have a border-based period 3, since $X[1:9] \not\approx X[4:12]$. X has a border-based period 6, since $X[1:6] \approx X[7:12]$.

Matsuoka et al. [39] proved analogous lemma to the periodicity lemma [26] for block-based periods. The periodicity lemma states that if a string X has two block-based periods p, q and $p + q - gcd(p, q) \leq |X|$, then $gcd(p, q)$ is also a block-based period of X.

Lemma 2.6 ([39]). *Let $\mathcal{P}$ be any non-empty set of positive integers with $gcd(\mathcal{P}) \notin \mathcal{P}$, and Σ be any alphabet of size at least 2. If a string $x \in \Sigma^*$ has block-based periods p for all $p \in \mathcal{P}$ and $|x| \geq b_{opt}(\mathcal{P})$, then x has also a block-based period $gcd(\mathcal{P})$.*

For exact matching, if $\mathcal{P} = \{p, q\}$, then $b_{opt}(\mathcal{P}) = p + q - gcd(p, q)$. For order-preserving matching, Matsuoka et al. [39] have also shown that $b_{opt}(\mathcal{P}) = p+q-gcd(p, q)$.

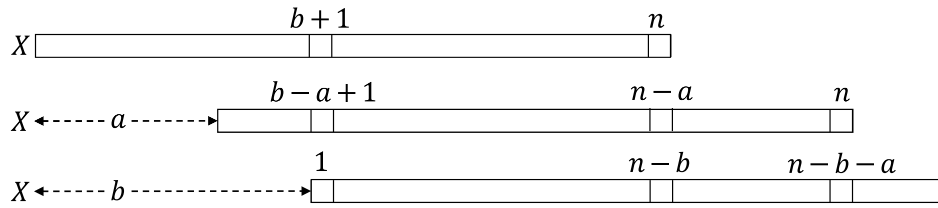


Figure 2.1: Suppose that a and b are periods of X. If $a+b < |X|$, then $(b+a)$ is a period of X. If $a < b$, then $(b-a)$ is a period of $X[1 : |X| - a]$.

Unfortunately, the analogous to the periodicity lemma does not hold for SCER border-periods. For instance, consider string $X = abacadaeafad$ in parameterized matching, it has a border-based periods $6, 8$, but does not have a border-based period of 2. The following lemma implies that if p is a period of X, then so is qp for every positive integer $q \leq \lfloor (|X| - 1)/p \rfloor$.

Lemma 2.7. *Suppose that a and b are periods of X. If $a + b < |X|$, then $(b + a)$ is a period of X. If $a < b$, then $(b - a)$ is a period of $X[1 : |X| - a]$.*

Proof: Let $n = |X|$. Since a is a period of X, by the definition $X[1:n-a] \approx X[1+a:n]$. Thus, $X[1 + b : n - a] \approx X[a + b + 1 : n]$. Similarly, since b is a period of X, by the definition $X[1 : n - b] \approx X[1 + b : n]$. Thus, $X[1 : n - b - a] \approx X[1 + b : n - a]$. Thus, $X[1 + b : n - a] \approx X[a + b + 1 : n] \approx X[1 : n - b - a]$, which means that $(b + a)$ is a period of X.

Since a and b are period of X, $X[1+b-a:n-a] \approx X[1+b:n]$ and $X[1:n-b] \approx X[1+b:n]$. Thus, by the transitivity property, $(b - a)$ is a period of $X[1 : n - a]$. ■

2.3 The model of parallel computation

In this thesis we will use Parallel Random Access Machine (PRAM) to model our algorithms. PRAM is a shared memory model, which neglects any hardware constraints. The processors work synchronously and communicates through the common random-access memory. The overhead of synchronizing the processors is also neglected. The cost of

arithmetic procedures such as addition, subtraction or check for an equality is considered to take a constant time. Parallel programs quickly become complicated, as we start to impose restrictions on the hardware. From the theoretical point of view, the simplicity and the universality of PRAM makes it convenient to assess efficiency of algorithms. It has been shown that PRAM can be simulated on more realizable parallel models with a polylogarithmic time cost [22].

The processors are indexed using natural numbers and execute the same program. Although they are executing the same program, each processors might be accessing different portions of the data. In one operation each processor can access one memory location. Depending on the allowance of simultaneous reads from or writes into the same memory location, PRAMs are divided into the four categories: Exclusive/Concurrent Read and Exclusive/Concurrent Write PRAMs. In this thesis, we will use the Concurrent Read Concurrent Write PRAM (CRCW-PRAM), which allows simultaneous reads and writes into the same memory location.

There are different policies for resolving conflicts during simultaneous writes. To avoid overcomplicating our algorithm with technical details we choose the Priority CRCW-PRAM. In the Priority CRCW-PRAM, When multiples processors try to write into the same memory location, the processor with the smallest index succeeds. It can be shown that Priority CRCW-PRAM can be simulated on the Common CRCW-PRAM, which is the weakest model among CRCW-PRAM, with a logarithmic overhead. In the Common CRCW-PRAM, the memory location is updated only if all processors that attempt to write the same value.

Algorithms modeled using the PRAM are assessed on two scales: time and work. Time is the time complexity of the algorithm, while work is the total number of elementary operations. There is a general theorem, called Brent's theorem, that relates the number of processors to the time and the work. The proof of Brent's theorem can be found in many literatures, but we present it as it is stated in [20].

Theorem 2.8 (Brent's theorem [10]). *Any synchronous parallel algorithm taking time t that consists of a total of x elementary operations can be implemented by p processors within a time of $\lfloor x/p \rfloor + t$.*

Designers of parallel algorithm strive for an algorithm, whose work complexity is same as the time complexity of the fastest know serial algorithm that solves the same problem.

Chapter 3

Serial duel-and-sweep algorithms for SCER

In this chapter we describe our serial duel-and-sweep pattern matching algorithm for SCER. At the end of the chapter, we discuss these algorithms on the instances of parameterized and order-preserving matchings. For our serial algorithm, it suffices to use $\approx$-prefix encoding. In this chapter, $\widetilde{X}$ means an $\approx$-prefix encoding for a string X, unless otherwise stated. For a string X of length n, we assume that $\widetilde{X}$ can be computed in $\tau_{|X|}$ time in serial. Assuming that $\widetilde{X}$ has been computed, we assume that re-encoding the element at position k with respect to some suffix $X[i:]$ takes $\xi_{|X[i:]|}$ time in serial.

3.1 Pattern matching for SCER

3.2 Overview of duel-and-sweep algorithm

We give an overview of the duel-and-sweep algorithm [2, 44], which is applicable to all our algorithms. The duel-and-sweep algorithm preprocesses the pattern first to find occurrences of the pattern inside the text efficiently. The pattern is first preprocessed to obtain a *witness table*, which is later used to prune candidates during the pattern search-

ing. As the name suggests, in the duel-and-sweep algorithm, the pattern searching is divided into two stages: the *dueling stage* and the *sweeping stage*. The pattern searching algorithm prunes candidates that cannot be pattern occurrences, first, by performing "duels" between, then by "sweeping" through the remaining candidates to obtain pattern occurrences.

First, we explain the idea of dueling. Suppose P is superimposed on itself with an offset $a < m$ and the two overlapped regions of P does not match. Then it is impossible for two candidates with offset a to match P. The dueling stage lets each pair of candidates with such offset a "duel" and eliminates one based on this observation, so that if candidate T_x gets eliminated during the dueling stage, then $T_x \not\approx P$. However, the opposite does not necessarily hold true: T_x surviving the dueling stage does not mean that $T_x \not\approx P$. On the other hand, it is guaranteed that if distinct candidates that survive the dueling stage overlap, their prefixes of certain length match. The sweeping stage takes the advantage of this property when checking whether some surviving candidate and the pattern match, so that this stage can be done also quickly.

Prior to the dueling stage, the pattern is preprocessed to construct a *witness table* based on which the dueling stage decides which pair of overlapping candidates should duel and how they should duel. For each offset $0 < a < m$, when the overlapped regions obtained by superimposing P on itself with offset a do not match, we need only one position i to say that the overlapping regions do not match. Given an offset a, w is a *witness for the offset* a for that offset if $\widetilde{P}_{a+1}[w] \neq \widetilde{P}[w]$ (Figure 3.1). We denote by $\mathcal{W}_P(a)$ the set of all witnesses for offset a. Obviously, $\mathcal{W}_P(a) = \emptyset$ for $a = 0, m-1, m$. The *witness table* $W[0 : m-1]$ is an array, such that $W[a]$ is a witness for offset a. When the overlap regions match for offset a, which implies that no witness exists for a, we express it as $W[a] = 0$.

More formally, in the dueling stage, we "duel" positions x and $x+a$ such that $\mathcal{W}_P(a) \neq \emptyset$ (see Figure 3.2). If $w \in \mathcal{W}_P(a)$, then it holds that

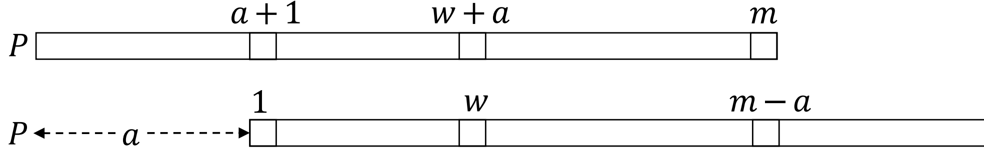


Figure 3.1: Illustration of a witness w for an offset a.

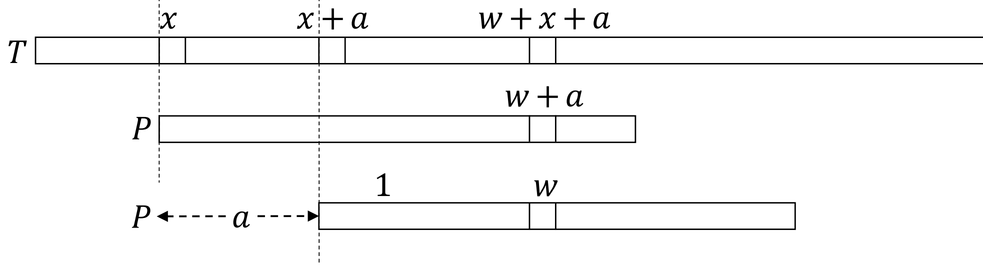


Figure 3.2: Candidate positions x and $x + a$ are performing a duel using a witness w for offset a.

- if $\widetilde{T}_{x+a}[w] = \widetilde{P}[w]$, then $T_x \not\approx P$,
- if $\widetilde{T}_{x+a}[w] \neq \widetilde{P}[w]$, then $T_{x+a} \not\approx P$.

Based on this observation, we can safely eliminate either candidate T_x or T_{x+a} without looking into other positions. This process is called *dueling* (Algorithm 1).

On the other hand, if the offset a has no witness pair, i.e. $P[1 : m - a] \approx P[a + 1 : m]$, no dueling is performed on them. We say that a position x *is consistent with* $x + a$ if $\mathcal{W}_P(a) = \emptyset$. The following lemma shows that the consistency property is transitive.

Lemma 3.1. *For any a, b, c such that $0 < a \leq b \leq c < m$, if a is consistent with b and b is consistent with c, then a is consistent with c.*

Algorithm 1: Dueling with respect to S. There is one survivor assuming x is not consistent with y.

```
1 Function Dueling(S̃, x, y)
2     w ← W[y − x];
3     if S̃_y[w] = P̃[w] then return y ;
4     return x;
```

Proof: Recall that a being consistent with b means that $b - a$ is a period of P. Then the fact that $b - a$ and $c - b$ are periods implies $c - a$ is a period by Lemma 2.7. ■

After the dueling stage, all surviving candidate positions are pairwise consistent. The dueling stage algorithm makes sure that no occurrence gets eliminated during the dueling stage. Taking advantage of the fact that surviving candidates from the dueling stage are pairwise consistent, the sweeping stage prunes them until all remaining candidates match the pattern. By ensuring pairwise consistency of the surviving candidates, the pattern searching algorithm reduces the number of times a position in the text is referenced during the sweeping stage.

3.2.1 Pattern preprocessing

The goal of the preprocessing stage is to compute a witness table $W[0{:}m]$, where $W[a] = 0$ iff $\mathcal{W}_P(a) = \emptyset$, otherwise $W[a] = w \in \mathcal{W}_P(a)$. First, we compute $LCP_P(a)$ for all $0 \le a < m$[1]. The procedure is described in Algorithm 2. Algorithm 2 has the following invariant.

- Variable a holds the leftmost offset $i > 0$ such that the value of $i + LCP[i]$ is the greatest.
- For all $i \in \{0, \dots, m-1\}$, $P[i+1 : i + LCP[i]] \approx P[1 : LCP[i]]$.

We discuss the correctness of Algorithm 2. The algorithm maintains variable a such that $P[a+1 : a+LCP[a]] \approx P[1 : LCP[a]]$ such that $(a+LCP[a])$ is as large as possible. For each $1 \le i < m$, the algorithm checks whether $i+1 \le a+LCP[a]$ and if so, since $P[1+a : a+LCP(a)] \approx P[1 : LCP(a)]$, we can say that $LCP[i] \ge \min(a+LCP[a]-i, LCP[i-a])$ (Figure 3.3); otherwise $LCP[i]$ is set to 0. At this point, the actual $LCP_P(i)$ may be larger. Thus, in the while loop, $LCP[i]$ is incremented until $P[1 : LCP[i]] \approx P[i+1 : i+Z_P[i]]$ and $P[1 : LCP[i]+1] \not\approx P[i+1 : i+LCP[i]+1]$. Now, by the condition (4) of Defintion 2.3,

[1] Algorithm for the construction of $LCP_P(a)$ for all $0 \le a < m$ is similar to Z-algorithm [28].

Algorithm 2: Serial algorithm for computing $LCP_P(i)$ for all $0 \leq i < m$

```
1  Function ComputeLCP(P̃)
2      Create array LCP[0 : m − 1] and initialize all elements to 0;
3      LCP[0] ← m;
4      a ← 1;
5      for i = 1 to m − 1 do
6          if i < a + LCP[a] then
7              LCP[i] ← min(a + LCP[a] − i, LCP[i − a]);
8          while i + LCP[i] ≤ m and P̃_{i+1}[LCP[i]] = P̃[LCP[i]] do
9              LCP[i] ← LCP[i] + 1;
10         if i + LCP[i] > a + LCP[a] then
11             a ← i;
12     return LCP;
```

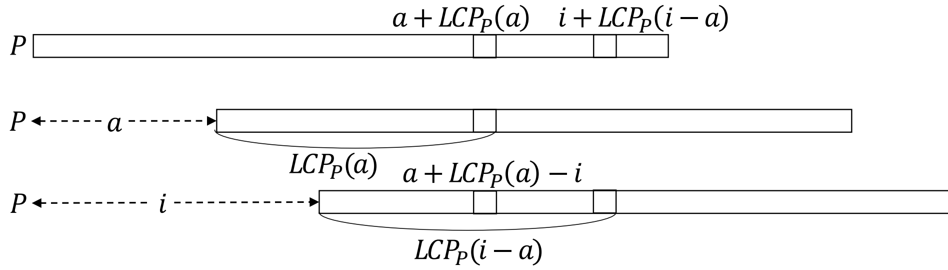


Figure 3.3: By Algorithm 2 invariant, $P[a+1 : a+LCP_P(a)] \approx P[1 : LCP_P(a)]$ and $P[i-a+1 : i-a+LCP_P(i-a)] \approx P[1 : LCP_P(i-a)]$. Thus, $LCP_P(i) \geq \min(a + LCP_P(a) - i, LCP_P(i-a))$.

$X_1 \not\approx Y_1 \implies X_1X_2 \not\approx Y_1Y_2$, where $X_1, X_2, Y_1, Y_2 \in \Sigma^*$ and X_1X_2 is a concatenation of X_1 and X_2 (similarly, Y_1Y_2 is a concatenation of Y_1 and Y_2). As a consequence of this, we have the following. If $P[i+1 : i+LCP[i]+1] \not\approx P[1 : LCP[i]+1]$, then for all the values $j > LCP[i]$, $P[i+1 : j] \not\approx P[1 : j]$.

Lemma 3.2. *Given $\widetilde{P}$, Algorithm 2 correctly computes $LCP_P(i)$ for all $0 \leq i < m$ in $O(\xi_m \cdot m)$ time.*

Proof: The condition in the while loop is checked in $O(\xi_m)$ time. The number of iterations of the outer loop is $O(m)$. Thus, the overall time complexity is $O(\xi_m \cdot m)$. ■

Using the value of $LCP_P(a)$, we can easily verify whether $\mathcal{W}_P(a)$ is empty or not. If

Algorithm 3: Serial algorithm for the pattern preprocessing

```
1 Function PreprocessingSerial(P̃)
2     create array of integers W[0 : m − 1];
3     LCP ← ComputeLCP(P̃);
4     for a = 0 to m − 1 do
5         if LCP[a] = m − a then W[a] ← 0;
6         else
7             W[a] ← LCP[a] + 1;
8     return W;
```

$LCP_P(a) = m - a$, that is, if P is overlapped on itself with the offset a and the overlapping regions match, then $\mathcal{W}_P(a) = \emptyset$. If $LCP_P[a] < m - a$, then $LCP_P(a) + 1$ is the tight mismatch position, i.e., $(LCP_P(a) + 1) \in \mathcal{W}_P(a)$.

Lemma 3.3. *For a pattern P of length m, Algorithm 3 constructs a witness table W_P in $O(\xi_m \cdot m)$ time, assuming that $\widetilde{P}$ is already computed.*

3.2.2 Pattern searching

Recall that the pattern searching consists of the dueling and the sweeping stages. The process of the dueling stage is shown in Algorithm 4. This stage eliminates candidates until all surviving candidate positions are pairwise consistent. The serial algorithm uses a stack to maintain candidates which are consistent with each other. A new candidate y will be pushed to the stack if the stack is empty. Otherwise y is checked by comparing it to the topmost element x of the stack. By the consistency transitivity, if x is consistent with y, all the other elements in the stack are consistent with y, too. Thus we can push y to the stack. On the other hand, if x is not consistent with y, we should exclude one of the candidates by dueling them. If x wins the duel, we put x back to the stack, discard y, and get a new candidate. If y wins the duel, we exclude x and continue comparison of y with the top element of the stack unless the stack is empty.

Lemma 3.4 ([2]). *The dueling stage can be done in $O(\xi_m \cdot n)$ time by using a witness*

Algorithm 4: Serial algorithm for the dueling stage

```
1  Function DuelingStageSerial(T̃, P̃, W)
2      create stack;
3      for y = 1 to n − m + 1 do
4          while stack is not empty do
5              pop x from stack;
6              if W[y − x] = 0 then
7                  push x and y to stack;
8                  break;
9              else
10                 surv ← Dueling(x, y − x);
11                 if surv = x then
12                     push x to stack;
13                     break;
14                 else
15                     push y to stack;
16                     break;
17         if stack is empty then
18             push y to stack;
19     return stack;
```

table W.

A naive implementation of sweeping requires $O(\xi_m \cdot n^2)$ time. Algorithm 5 takes advantage of the fact that all the remaining candidates are pairwise consistent, so that we can reduce the time complexity to $O(\xi_m \cdot n)$ time. Suppose there is a tight mismatch at position j when comparing P with T_x, that is, $T_x[1{:}j{-}1] \approx P[1{:}j{-}1]$ and $T_x[1{:}j] \not\approx P[1{:}j]$. If the next candidate is T_{x+a} with $a < j$, since $P[1 : j - a - 1] \approx P[a + 1 : j - 1] \approx T_x[a + 1 : j - 1] \approx T_{x+a}[1 : j - a - 1]$, we can start comparison of P and T_{x+a} from the position where the mismatch with T_x occurred. If $P \approx T_x$, the above discussion holds for $j = m + 1$. Therefore, the total number of comparison is bounded by $O(n)$, by applying the same argument on the complexity of the KMP algorithm for exact matching.

Lemma 3.5. *The sweeping stage can be completed in $O(\xi_m \cdot n)$ time.*

By Lemmas 3.3, 3.4, and 3.5, we summarize this section as follows.

Algorithm 5: Serial algorithm for the sweeping stage

```
1  Function SweepingStageSerial(P̃, T̃)
2      while there are unchecked candidates to the right of T_x do
3          let T_x be the leftmost unchecked candidate;
4          if there are no candidates overlapping with T_x then
5              if T_x ≉ P then eliminate T_x;
6          else
7              let T_{x+a} be the leftmost candidate that overlaps with T_x;
8              if T_x ≈ P then start checking T_{x+a} from the position m − a + 1 ;
9              else
10                 let j be the mismatch position;
11                 eliminate T_x;
12                 start checking T_{x+a} from the position j − a;
```

Theorem 3.6. *Given a text T of length n and a pattern P of length m, the duel-and-sweep algorithm solves the pattern matching problem for a SCER $\approx$ in $O(\tau_n + \xi_m \cdot n)$ time.*

In the following sections we discuss specific instances of SCER. Since prev-encoding for parameterized matching and parent-distance encoding for cartesian-tree matching are $\approx$-encodings, algorithms for solving the pattern matching problem for these matchings directly from the SCER algorithm. Nearest-neighbor encoding for order-preserving matching and encoding for parameterized matching are not $\approx$-encodings. Despite this fact, we show that there exists a duel-and-sweep algorithm similar to the SCER algorithm that solves the pattern matching problem for these SCER. For order-preserving matching, we further extend the algorithm to solve the 2D pattern matching.

3.3 Parameterized matching

We fix two alphabets Σ and Π. We call elements of Σ *constant* symbols and those of Π *parameter* symbols. An element of Σ^* is called a *constant string* and that of $(\Sigma \cup \Pi)^*$ is called a *parameterized string*, or *p-string* for short. We assume that the size of Σ and Π

are constant. Two strings X and Y of the same length *parameterized match*, if X can be transformed into Y by applying a bijection g between the characters of X and Y.

Definition 3.7 (Parameterized equivalence [5]). *Given two p-strings X and Y of length n, $X \approx Y$, iff there is a bijection f on $\Sigma \cup \Pi$ such that $f(a) = a$ for any $a \in \Sigma$ and $f(X[i]) = Y[i]$ for all $1 \leq i \leq n$ [5].*

We can determine whether $X \approx Y$ or not by using a $\approx$-encoding called *prev-encoding* defined as follows.

Definition 3.8 (Prev-encoding [5]). *For a string X of length n over $\Sigma \cup \Pi$, the* prev-encoding *for X, denoted by $prev(X)$, is defined to be a string over $\Sigma \cup \mathcal{N}$ of length n such that for each $1 \leq i \leq n$,*

$$prev(X)[i] = \begin{cases} X[i] & \textit{if } X[i] \in \Sigma, \\ \infty & \textit{if } X[i] \in \Pi \textit{ and } X[i] \neq X[j] \textit{ for } 1 \leq j < i, \\ i - k & \textit{if } X[i] \in \Pi \textit{ and } k = \max\{j \mid X[j] = X[i] \textit{ and } 1 \leq j < i\}. \end{cases} \quad (3.1)$$

For any strings X and Y, $X \approx Y$ if and only if $prev(X) = prev(Y)$ [5]. For example, given $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Pi = \{\mathtt{u}, \mathtt{v}, \mathtt{x}, \mathtt{y}\}$, $X = \mathtt{uvvvauuvb}$ and $Y = \mathtt{xyyyaxxyb}$ are p-matches by f such that $f(\mathtt{u}) = \mathtt{x}$ and $f(\mathtt{v}) = \mathtt{y}$, where $prev(X) = prev(Y) = 0011\mathtt{a}514\mathtt{b}$. It is easy to see that prev-encoding is a $\approx$-encoding. We denote the prev-encoding of suffix $X[k:]$ as $prev(X)_k$. For instance, for $X = \mathtt{uvvvauuvb}$, $prev(X)_3 = 01\mathtt{a}014\mathtt{b}$.

Lemma 3.9. *([5]) Given a string X, $prev(X)$ can be constructed in $O(|X| \log |\Pi|)$ time in serial.*

Given $prev(X)[i]$, $prev(X)_k[i]$ can be computed in the following manner in $O(1)$ time.

$$prev(X[x:n])[i] = \begin{cases} \infty & \text{if } X[x+i-1] \in \Pi \text{ and } prev(X)[x+i-1] \geq i, \\ prev(X)[x+i-1] & \text{otherwise.} \end{cases} \tag{3.2}$$

From the discussions above, for parameterized matching $\tau_n = O(n \log |\Pi|)$ and $\xi_m = O(1)$. By simple extension of the SCER algorithm described in the previous section, we obtain the following the result.

Theorem 3.10. *Parameterized pattern matching problem can be solved in $O(n \log |\Pi| + n)$ time in serial, where $|\Pi|$ is size of the variable alphabet.*

3.4 Cartesian-tree matching

A string X can be associated with its corresponding Cartesian tree CT_X according to the following rules [41]:

- If X is an empty string, CT_X is an empty tree.
- If $X[1:n]$ is not empty and $X[i]$ is the minimum value, CT_X is the tree with $X[i]$ as the root, $CT_{X[1:i-1]}$ as the left subtree, and $CT_{S[i+1..n]}$ as the right subtree. If there are two or more minimum values, we choose the leftmost one as the root.

Two strings X and Y cartesian-tree match iff $CT_X = CT_Y$. For a string X, its parent-distance encoding [41] for cartesian-tree matching PD_X is defined as follows.

$$PD_X[i] = \begin{cases} i - \max_{1 \leq j < i}\{j \mid X[j] \leq X[i]\} & \text{if such } j \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

Given PD_X, $PD_{X[x:n]}[i]$ can be computed in the following manner in $O(1)$ time.

$$PD_{X[x:n]}[i] = \begin{cases} 0 & \text{if } PD_X[x+i-1] \geq i, \\ PD_X[x+i-1] & \text{otherwise.} \end{cases}$$

Lemma 3.11. *([41]) Given a string X of length n, PD_X can be computed in $O(n)$ time in serial.*

Parent-distance encoding is an $\approx$-prefix encoding, thus, by simple extension of the SCER algorithm we have the following.

Theorem 3.12. *Cartesian-tree pattern matching problem can be solved in $O(n)$ time in serial.*

3.5 Palindrome matching

Palindrome is a string that reads same both in forward and backward direction. Let Σ be a finite alphabet. For a string $X \in \Sigma^*$, let X^R be the reverse of X. A string X is a *palindrome* iff $X = X^R$. If $|X|$ is even, then $X = YY$ for some $Y \in \Sigma^*$. If $|X|$ is odd, then $X = YyY$ for some $Y \in \Sigma^*$ and $y \in \Sigma$. The *radius* of a palindrome X is equal to $|X|/2$.

Given a string X, a palindromic *center* of a substring $X[i : j]$ is $(i + j)/2$. A palindromic substring $X[i : j]$ is called the *maximal palindrome* at center $(i + j)/2$ if no other palindrome at center $(i + j)/2$ has a radius larger than $X[i : j]$. Let $Pals(X)$ be an ordered set of pairs (c, r) where r is the radius of the maximal palindrome at centers $c = 1, 1.15, 2, \ldots, |X|$. Formally, $Pals_X = \{(c, r) \mid X[c - r + 0.5 : c + r - 0.5]$ is a maximal palindrome at center $c = 1, 1.5, 2, \ldots, |X|\}$. For the rest of the thesis, we assume that $Pals_X$ is sorted in increasing order of centers and we denote the radius of the maximal palindrome at center c as $Pals_X(c)$. Also, the we denote the left and right

ends of the maximal palindrome of X centered at c as $L_X(c)$ and $R_X(c)$, respectively. That is, $L_X(c) = c - Pals_X(c) + 0.5$ and $R_X(c) = c + Pals_X(c) - 0.5$. Given two p-strings X and Y of length n, X and Y *palindrome match (pal-match)*, if $Pals_X = Pals_Y$ [42].

Lemma 3.13. *([38]) Given a string X, $Pals_X$ can be constructed in $O(|X|)$ time in serial.*

Given $Pals_X$, $Pals_{X[x:y]}(c)$ such that $x \leq c \leq y$ can be computed in the following manner in $O(1)$ time. Let $c' = c + x - 1$.

$$Pals_{X[x:y]}(c) = \begin{cases} Pals_X(c') & \text{if } x \leq L_X(c') \text{ and } R_X(c') \leq y, \\ \min(c' - x + 0.5, y - c' + 0.5) & \text{otherwise.} \end{cases} \tag{3.3}$$

For a string X, to efficiently compute $LCP_X(i)$ for all $0 \leq i < |X|$, we define array $Lpal_X$ [42]. The value of $Lpal_X[i]$ is the length of the longest palindrome that ends at position i of X. Formally, for a string X of length n, $Lpal_X$ is an integer array of length n such that $Lpal_X[i] = \max\{i - k + 1 \mid X[k : i] = X[k : i]^R, 1 \leq k \leq i\}$.

Lemma 3.14. *([42]) For two strings of same length X and Y, $Pals_X = Pals_Y$ iff $Lpal_X = Lpal_Y$.*

Tomohiro et al. [42] showed that, given $Pals_X$, $Lpal_X$ can be constructed in $O(|X|)$ time in online fashion. $Pals_X$ is not an $\approx$-prefix encoding, however $Lpal_X$ is an $\approx$-prefix encoding. Given $Pals_P$ and $Lpal_P$, computing $Lpal_{P[s:i]}[i - s + 1]$ for random s and i takes $O(m)$ time. First, we show that there exists an $O(m)$ time algorithm that computes $LCP_P(i)$ for all $0 \leq i < m$, given $Lpal_P$. On Line 8 of Algorithm 2, the SCER algorithm checks if $\widetilde{P}_{i+1}[LCP[i]] = \widetilde{P}[LCP[i]]$. This, this is equivalent to checking for $P[1 + i : i + LCP[i]] \approx P[1 : LCP[i]]$ or not, assuming that $P[1 + i : i + LCP[i] - 1] \approx P[1 : LCP[i] - 1]$. Thus, we need to know the value of $Lpal_{P[s:i]}[i - s + 1]$ for some s and i.

We define *active center* for $Lpal_{P[s:i]}$. By the definition $Lpal_{P[s:i]}[i - s + 1] = 2(i - c) + 1$ where (c, r) is the maximal palindrome in $Pals_P$ such that c is the smallest center satisfying

Algorithm 6: Serial algorithm for computing $LCP_P(i)$ for all $0 \leq i < m$ (palindrome matching)

```
1  create global variable c;
2  c ← 0;
3  Function GetLpal(i, x)
4      Let (c − i, r) ∈ Pals_{P[i+1:i+x]};
5      while c − i + r < x do
6          c ← c + 0.5;
7          Let (c − i, r) ∈ Pals_{P[i+1:i+x]};
       // p = Lpal_{P[i+1:i+x]}[x]
8      lpal ← 2(x − (c − i)) + 1;
9      return lpal;
10 Function ComputeLCP(Pals_P, Lpal_P)
11     Create array LCP[0 : m − 1] and initialize all elements to 0;
12     LCP[0] ← m;
13     a ← 1;
14     for i = 1 to m − 1 do
15         if i < a + LCP[a] then
16             LCP[i] ← min(a + LCP[a] − i, LCP[i − a]);
17         while i + LCP[i] ≤ m and GetLpal(i, LCP[i]) = Lpal_P[LCP[i]] do
18             LCP[i] ← LCP[i] + 1;
19         if i + LCP[i] > a + LCP[a] then
20             a ← i;
21     return LCP;
```

$c \geq (s+i)/2$ and $c+r \geq i$. We call such center c the *active center* and denote is $AC_P(s,i)$. Thus, $Lpal_{P[s:i]}[i-s+1] = 2(i - AC_P(s,i)) + 1$. Tomohiro et al. [42]showed that if s and i monotonically increase from 1 to m, then the total cost for computing $Lpal_{P[s:i]}[i-s+1]$ for all s and i never exceeds the number of the centers in P, which is $2m-1$.

Lemma 3.15. *([42]) Given a string X be any string of length n, for any integers s, i, s', i' such that $1 \leq s \leq i \leq n$ and $1 \leq s' \leq i' \leq m$, if $s \leq s'$ and $i \leq i'$, then $AC_X(s,i) \leq AC_X(s',i')$.*

The algorithm for computing $LCP_P(i)$ for all $0 \leq i < m$ is shown in Algorithm 6. `ComputeLCP` of Algorithm 6 is similar to Algorithm 2, except for the while-loop condition check. `GetLpal`(i,x) returns the value of $Lpal_{P[i+1:i+x][x]}$. Algorithm 6 has the following

invariant in addition to the invariants of the SCER algorithm (Algorithm 2).

- At the end of each iteration of the outer for loop, the global variable c holds the value of $AC_P(i+1, i+LCP_P[i])$.

We discuss the correctness of Algorithm 6. Suppose that Algorithm 6 correctly computed $LCP_P(j)$ for all $0 < j < i$ and $LCP[i-1]$ is equal to $LCP_P(i-1)$, we have the prove that the algorithm correctly computes $LCP_P(i-1)$. On Line 15, the value of $LCP[i]$ is set to $\min(a+LCP[a]-i, LCP[i-a])$. Since, for a, $P[a+1 : a+LCP[a]] \approx P[1 : LCP[a]]$ such that $(a+LCP[a])$ is as large as possible, $i+LCP[i] \geq j+LCP[j]$ for any $0 < j < i$. Using Lemma 3.15, $AC_P(i+1, i+LCP[i]) \geq AC_P(j+1, j+LCP[j])$ for any $0 < j < i$. Thus, $AC_P(i+1, i+LCP[j]) \geq c$, and `GetLpal` naively searches for $AC_P(i+1, i+LCP[j])$ starting from center c. Now, given integers i a, let us consider finding $Lpal_{P[i+1:i+x]}[x]$. By Lemma 3.15, $AC_P(i+1, i+x-1) \leq AC_P(i+1, i+x)$. Before the algorithm calls `GetLpal` for i and x, the value of c is set to $AC_P(i+1, i+x-1)$ and algorithm searches for $AC_P(i+1, i+x)$ starting from center c.

Lemma 3.16. *Given $Pals_P$ and $Lpal_P$, Algorithm 6 correctly computes $LCP_P(i)$ for all $0 \leq i < m$ in $O(m)$ time.*

Proof: By Lemma 3.15, to compute $LCP_P(i)$ for all $0 < i < m$, each center is considered $O(1)$ times and there are $O(m)$ centers. Since, for i and x, $(c-i, r) \in Pals_{P[i+1:i+x]}$ can be computed in $O(1)$, the overall time complexity is $O(m)$. ■

Since $Lpal_P$ is an $\approx$-prefix encoding, by simple extension of the SCER algorithm, we obtain a $O(n)$ pattern searching algorithm.

Theorem 3.17. *Palindrome pattern matching problem can be solved in $O(m+n)$ time in serial.*

3.6 Order-preserving matching

3.6.1 One-dimensional matching

We say that two strings X and Y of equal length n are *order-isomorphic*, written $X \approx Y$, if

$$X[i] \leq X[j] \Longleftrightarrow Y[i] \leq Y[j] \quad \text{for all } 1 \leq i, j \leq n.$$

For instance, $(12, 35, 5) \approx (25, 30, 21) \not\approx (11, 13, 20)$. If $X \not\approx Y$, then, there must exist a pair $\langle i, j \rangle$ of positions ($i < j$) such that the condition above does not hold. We will call such $\langle i, j \rangle$ a *mismatch position pair* for X and Y. We say that a mismatch position pair $\langle i, j \rangle$ is *tight* if $X[1 : j-1] \approx Y[1 : j-1]$ and $X[1 : j] \not\approx Y[1 : j]$.

In order to check the order-isomorphism of a string X with another string, Kubica et al. [37] defined useful arrays $Lmax_X$ and $Lmin_X$ by

$$Lmax_X[i] = j \quad \text{if} \ \ X[j] = \max_{k<i}\{X[k] \mid X[k] \leq X[i]\},$$
$$Lmin_X[i] = j \quad \text{if} \ \ X[j] = \min_{k<i}\{X[k] \mid X[k] \geq X[i]\}.$$

We use the rightmost (largest) j if there exist more than one such $j < i$. If there is no such j then we define $Lmin_X[i] = 0$ and $Lmax_X[i] = 0$. We will refer to $Lmax_X$ and $Lmin_X$ as a *nearest neighbor encoding* for order-preserving matching. It is easy to verify that the nearest neighbor encoding is a $\approx$-prefix encoding.

Lemma 3.18 ([37]). *For a string X, $Lmax_X$ and $Lmin_X$ can be computed in $O(|X| \log |X|)$ time.*

From the definition, we can easily observe the following properties. For a position

$i \in \{1, \ldots, |X|\}$ such that $Lmax_X[i] \neq 0$ and $Lmin_X[i] \neq 0$,

$$X[Lmax_X[i]] = X[i] \iff X[i] = X[Lmin_X[i]], \tag{3.4}$$

$$X[Lmax_X[i]] < X[i] \iff X[i] < X[Lmin_X[i]]. \tag{3.5}$$

Using $Lmax_X$ and $Lmin_X$, the order-isomorphism of two strings can be decided as follows. Assuming that $Lmax_X$ and $Lmin_X$ are computed, for two strings X and Y of same length, we define

$$F(X, Y, i) = \begin{cases} i_{\min} & \text{if } i_{\min} \neq 0 \text{ and } Y[i_{\min}] < Y[i], \\ i_{\max} & \text{if } i_{\max} \neq 0 \text{ and } Y[i_{\max}] > Y[i], \\ 0 & \text{otherwise,} \end{cases} \tag{3.6}$$

where $i_{\min} = Lmin_X[i]$ and $i_{\max} = Lmax_X[i]$. If both conditions in Equation 3.6 hols, either $i_{\min}$ or $i_{\max}$ can be taken. For $a = F(X, Y, i)$, if $a \neq 0$, then $\langle a, i \rangle$ is a mismatch position pair for $X \not\approx Y$. When checking order-isomorphism of two strings using the nearest neighbor encoding, it suffices to encode only one of them.

Lemma 3.19 ([16]). *For two strings X and Y of length n, assume that $X[1 : i-1] \approx Y[1 : i-1]$ for some $0 < i < n$. Then $X[1 : i] \approx Y[1 : i]$ iff $F(X, Y, i) = 0$.*

Corollary 3.20. *Given Two strings X and Y of length n, if for some position i, $F(X, Y, i) \neq 0$, then $X \not\approx Y$.*

For a string S, $Lmax_X$ and $Lmin_X$ arrays can be computed in $O(sort(X) + |X|)$ time.

Lemma 3.21 ([37]). *For a string X, let $sort(X)$ be the time required to sort the elements of S. $Lmax_X$ and $Lmin_X$ can be computed in $O(sort(X) + |X|)$ time.*

The nearest neighbor encoding is convenient for checking the order-isomorphism, but it is expensive to re-encode a suffix of a string. Thus, for our pattern matching algorithm

for order-preserving pattern matching (OPPM), we encode only the pattern, once, at the beginning, and work with the "raw" pattern and text. $Lmax_P$ and $Lmax_P$ will be used only for the following two situations.

- checking order-isomorphism of candidates with the pattern in the sweeping stage, and
- checking order-isomorphism of prefixes of P with their corresponding suffixes of P, when constructing a witness table.

Next, we modify the definition of witnesses for OPPM. For each offset $0 < a < m$, the SCER algorithm saves a single position i such that $\widetilde{P}_{a+1}[i] \neq \widetilde{P}[i+a]$. Such position i was called a witness for the offset a. Let us consider a case when P is not encoded. In OPPM, we need two positions as a witness to say that the two strings are not order-isomorphic. Therefore, when the overlapped regions obtained by superimposing P on itself with offset a are not order-isomorphic, we use a mismatch position pair $\langle i, j \rangle$ called *a witness pair for offset* a if either of the following holds:

- $P[i] = P[j]$ and $P[i + a] \neq P[j + a]$,
- $P[i] > P[j]$ and $P[i + a] \leq P[j + a]$,
- $P[i] < P[j]$ and $P[i + a] \geq P[j + a]$.

Thus, $W[a]$ contains a witness pair, if $\mathcal{W}_P(a) \neq \emptyset$. When the overlap regions are order-isomorphic for offset $\mathcal{W}_P(a) = \emptyset$, $W[a] = \langle 0, 0 \rangle$. Hereinafter, we will refer to $\langle 0, 0 \rangle$ as a *zero*. Table 3.1 shows an example of a witness table. We denote a pair of elements $P[i]$ and $P[j]$ in P as $P[i, j]$. If for $P[i, j]$ and $P[i + a, j + a]$ either of the above conditions hold, we write it as $P[i, j] \not\approx P[i + a, j + a]$.

Table 3.1: Witness table W for a string $P = (18, 22, 12, 50, 10, 17)$. For instance, the witness pair $W[2]$ for offset 2 is $\langle 2, 4\rangle$, because $P[2] = 22 < 50 = P[4]$ and $P[2+2] = 50 > 17 = P[4+2]$. On the other hand, $W[4] = \langle 0, 0\rangle$, since $P[1:2] \approx P[5:6]$.

	0	1	2	3	4	5	6
P		18	22	12	50	10	17
W	$\langle 0, 0\rangle$	$\langle 1, 2\rangle$	$\langle 2, 4\rangle$	$\langle 1, 2\rangle$	$\langle 0, 0\rangle$	$\langle 0, 0\rangle$	$\langle 0, 0\rangle$

Table 3.2: The Z-array of $P = (18, 22, 12, 50, 10, 17)$. For instance, $Z_P[3] = 3$ because $P[1:3] = (18, 22, 12) \approx (12, 50, 10) = P[3:5]$ and $P[1:4] = (18, 22, 12, 50) \not\approx (12, 50, 10, 17) = P[3:6]$. $Lmax_P$ and $Lmin_P$ are also shown.

	1	2	3	4	5	6
P	18	22	12	50	10	17
Z_P	6	1	3	1	2	1
$Lmax_P$	0	1	0	2	0	3
$Lmin_P$	0	0	1	0	3	1

Pattern preprocessing

Recall that the goal of the preprocessing stage is to compute the witness table $W[0:m]$, where $W[a] = \langle 0, 0\rangle$ iff $\mathcal{W}_P(a) = \emptyset$, otherwise $W[a] = \langle w_1, w_2\rangle \in \mathcal{W}_P(a)$. The preprocessing of our serial algorithm is described in Algorithm 3. First, we construct the $Lmax_P$ and $Lmin_P$ arrays for P. Hasan et al. [29] gave an algorithm to compute Z-array for OPPM. Recall that, for P, the *Z-array* of P for OP matching is defined by

$$Z_P[i] = \max_{1 \le j \le m-i+1} \{j \mid P[1:j] \approx P[i:i+j-1]\} \quad \text{for each } 1 \le i \le m.$$

An example of the Z-array is illustrated in Table 3.2.

Lemma 3.22. *([29]) For a string X, the Z-array Z_X can be computed in $O(|X|)$ time, assuming that $Lmax_X$ and $Lmin_X$ are already computed.*

Using the value of $Z_P[i]$, we can verify whether $\mathcal{W}_P(i-1)$ is empty or not. If $Z_P[i] = m - i + 1$, that is, if P is overlapped on itself with the offset $(i-1)$ and the overlapping regions are order-isomorphic, then $\mathcal{W}_P(i-1) = \emptyset$. If $Z_P[i] = j < m - i + 1$, then $\mathcal{W}_P(i-1) \neq \emptyset$ and there must exist a position pair $\langle j', j\rangle \in \mathcal{W}_P(i-1)$, where $j' < j$.

Algorithm 7: Dueling for OPPM

```
1 Function Dueling(x, a)
2     ⟨i, j⟩ ← W[a];
3     surv ← x + a;
4     if (P[i], P[j]) ≉ (T[x + a + i − 1], T[x + a + j − 1]) then
5         surv ← x;
6     return surv;
```

Specifically, for an offset a, let $i = Z_P[a+1]+1$. Suppose that $i_{\min} = Lmin_P[i]$ and $i_{\max} = Lmax_P[i]$.

- If $i_{\max} \neq 0$ and $(P[i_{\max}], P[i]) \not\approx (P[i_{\max}+a], P[i+a])$, then $\langle i_{\max}, i\rangle \in \mathcal{W}_P(a)$.
- If $i_{\min} \neq 0$ and $(P[i_{\min}], P[i]) \not\approx (P[i_{\min}+a], P[i+a])$, then $\langle i_{\min}, i\rangle \in \mathcal{W}_(a)$.

Lemma 3.23. *For a pattern P of length m, Algorithm 3 constructs a witness table W in $O(m)$ time assuming that Z_P is already computed.*

Pattern searching

In the dueling stage, witness pairs are used in the following manner. Suppose that $W[a] = \langle i, j\rangle$, where $P[i] < P[j]$ and $P[i+a] \geq P[j+a]$, for example. Then, it holds that

- if $T[x+a+i-1] \geq T[x+a+j-1]$, then $T_{x+a} \not\approx P$,
- if $T[x+a+i-1] < T[x+a+j-1]$, then $T_x \not\approx P$.

We can perform this processes similarly for other equality/inequality cases. Dueling for OPPM is described in Algorithm 7. Fig. 3.4 gives an example run of the dueling stage.

Lemma 3.24 ([2]). *The dueling stage can be done in $O(n)$ time by using W.*

Now, we discuss the sweeping stage for OPPM. Suppose that we need to check whether some surviving candidate T_x is order-isomorphic to P. It suffices to successively compute $F(P, T_x, i)$ in Equation 3.6, starting from the leftmost position in T_x. If $F(P, T_x, i) = 0$

		1	2	3	4	5	6	7	8	9	10	stack
$y = 1$	add T_1	8	13	5	21	14	18	20	25	15	22	1
$y = 2$	exclude T_2	8	13	5	21	14	18	20	25	15	22	1
		12	50	10	17							
			12	50	10	17						
$y = 3$	add T_3	8	13	5	21	14	18	20	25	15	22	1,3
$y = 4$	exclude T_4	8	13	5	21	14	18	20	25	15	22	1,3
				12	50	10	17					
					12	50	10	17				
$y = 5$	add T_5	8	13	5	21	14	18	20	25	15	22	1,3,5
$y = 6$	exclude T_5 add T_6	8	13	5	21	14	18	20	25	15	22	1,3,6
						12	50	10	17			
							12	50	10	17		
$y = 7$	exclude T_6 add T_7	8	13	5	21	14	18	20	25	15	22	1,3,7
							12	50	10	17		
								12	50	10	17	

Figure 3.4: An example run of the dueling stage for $T = (8, 13, 5, 21, 14, 18, 20, 25, 15, 22)$, $P = (12, 50, 10, 17)$, and $W = (\langle 0,0\rangle, \langle 1,2\rangle, \langle 0,0\rangle, \langle 0,0\rangle, \langle 0,0\rangle)$. First, the position 1 is pushed to the stack. Next, T_2 duels with T_1 and then T_2 loses because $P[1] < P[2]$ and $T_2[1] > T_2[2]$. The next position 3 is pushed to the stack by $W[3-1] = \langle 0,0\rangle$. Similarly, T_4 loses against T_3, and 5 is accepted to the stack. For $y = 6$, T_5 is removed and T_6 is added to the stack because $P[1] < P[2]$, $T_6[1] < T_6[2]$, and 3 is consistent with 6. Finally T_7 is defeats T_6 and the contents of the stack become 1, 3, and 7.

for all positions i in T_x, then $T_x \approx P$. Otherwise, $T_x \not\approx P$, and obtain a mismatch position j, such that $F(P, T_x, j) \neq 0$.

Suppose there is a mismatch at position j when comparing P with T_x, that is, $T_x[1 : j-1] \approx P[1 : j-1]$ and $T_x[1 : j] \not\approx P[1 : j]$. If the next candidate is T_{x+a} with $a < j$, since $P[1 : j-a-1] \approx P[a+1 : j-1] \approx T_x[a+1 : j-1] = T_{x+a}[1 : j-a-1]$, we can start comparison of P and T_{x+a} from the position where the mismatch with T_x occurred. If $P \approx T_x$, the above discussion holds for $j = m+1$. Therefore, the total number of comparison is bounded by $O(n)$. The analysis is same as that for the SCER algorithm.

Lemma 3.25. *The sweeping stage can be completed in $O(n)$ time.*

By Lemmas 3.3, 3.4, and 3.5, we summarize this section as follows.

Theorem 3.26. *Given a text T of length n and a pattern P of length m, the serial duel-and-sweep algorithm solves the OPPM problem in $O(n + m \log m)$ time. Moreover, the*

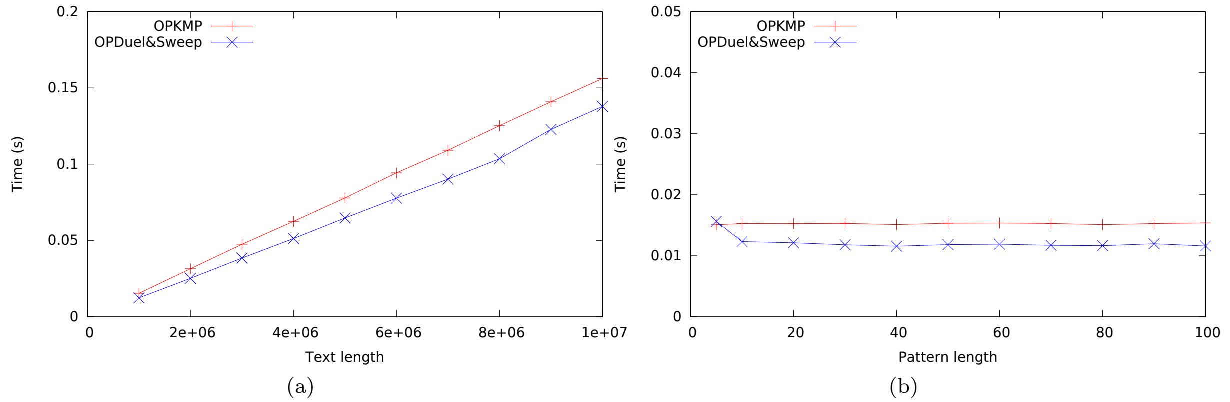


Figure 3.5: Running time of the algorithms with respect to (a) text length, and (b) pattern length.

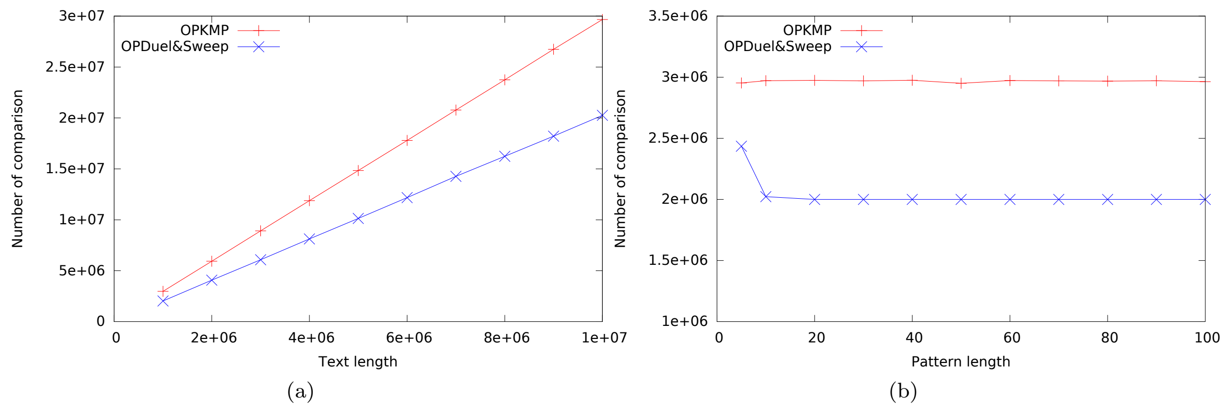


Figure 3.6: Number of comparisons in the algorithms with respect to (a) text length, and (b) pattern length.

running time is $O(n + m)$ under the natural assumption that the characters of P can be sorted in $O(m)$ time.

Experiments

In order to compare the performance of proposed algorithm with the KMP-based algorithm [37, 35] on solving the OPPM problem, we performed two sets of experiments. In the first experiment set, the pattern size m is fixed to 10, while the text size n is changed from 100000 to 1000000. In the second experiment set, the text size n is fixed to 1000000 while the pattern size m is changed from 5 to 100. We measured the average of running time and the number of comparisons for 50 repetitions on each experiment. We used

randomly generated texts and patterns with alphabet size $|\Sigma| = 1000$. Experiments are executed on a machine with Intel Xeon CPU E5-2609 8 cores 2.40 GHz, 256 GB memory, and Debian Wheezy operating system.

The results of our experiments are shown in Figs. 3.5 and 3.6. We can see that our algorithm is better than the KMP-based algorithm in running time and the number of comparisons when the pattern size and text size are large. However, our algorithm was slower when the pattern is very short, namely $m = 5$. The reason why the proposed algorithm makes fewer comparisons than the KMP-based algorithm may be explained as follows. The KMP-based algorithm relies on Lemma 3.19, which compares symbols at three positions to check the order-isomorphism between a prefix of the pattern and a substring of the text when the prefix is extended by one. On the other hand, the dueling stage of our algorithm compares only two positions determined by the witness table. By pruning candidates in the dueling phase, the number of precise tests of order-isomorphism in the sweeping stage is reduced.

3.6.2 Two-dimensional matching

In this section, we will discuss how to perform two-dimensional order preserving pattern matching (2d-OPPM). Array indexing is used for two-dimensional strings, the horizontal coordinate x increases from left to right and the vertical coordinate y increases from top to bottom. $\boldsymbol{S}[x, y]$ denotes an element of $\boldsymbol{S}$ at position (x, y) and $\boldsymbol{S}[x\!:\!x+w-1, y\!:\!y+h-1]$ denotes a substring of $\boldsymbol{S}$ of size $w \times h$ with top-left corner at the position (x, y).

We say that two dimensional strings $\boldsymbol{S}$ and $\boldsymbol{T}$ are *order-isomorphic*, written $\boldsymbol{S} \approx \boldsymbol{T}$, if $\boldsymbol{S}[i_x, i_y] \leq \boldsymbol{S}[j_x, j_y] \iff \boldsymbol{T}[i_x, i_y] \leq \boldsymbol{T}[j_x, j_y]$ for all $1 \leq i_x, j_x \leq w$ and $1 \leq i_y, j_y \leq h$. For a simple presentation, we assume that both text and pattern are squares ($w = h$) in this paper, but we can generalize it straightforwardly.

Definition 3.27 (2d-OPPM problem)**.** *The two-dimensional order-preserving matching problem is defined as follows,*

Input: *A text* $\boldsymbol{T}$ *of size* $n \times n$ *and a pattern* $\boldsymbol{P}$ *of size* $m \times m$*,*

Output: *All occurrences of substrings of* $\boldsymbol{T}$ *that are order-isomorphic to* $\boldsymbol{P}$*.*

Our approach is to reduce 2d-OPPM problem into 1d-OPPM problem, based on the following observation. For two-dimensional string $\boldsymbol{S}$, let *serial*($\boldsymbol{S}$) be a (one-dimensional) string which *serializes* $\boldsymbol{S}$ by traversing it in the left-to-right/top-to-bottom order. We can easily verify the following lemma.

Lemma 3.28. $\boldsymbol{S} \approx \boldsymbol{T}$ *if and only if serial*($\boldsymbol{S}$) $\approx$ *serial*($\boldsymbol{T}$) *for any* $\boldsymbol{S}$ *and* $\boldsymbol{T}$*.*

Theorem 3.29. *2d-OPPM problem can be solved in* $O(n^2 m + m^2 \log m)$*.*

Proof: For a fixed $1 \le x \le n-m+1$, consider the substring $\boldsymbol{T}[x:x+m-1, 1:n]$ and let $S_x = \mathit{serial}(\boldsymbol{T}[x:x+m-1, 1:n])$. By Lemma 3.28, $\boldsymbol{P}$ occurs in $\boldsymbol{T}$ at position (x, y), i.e. $\boldsymbol{P} \approx \boldsymbol{T}[x:x+m-1, y:y+m-1]$ if and only if $\mathit{serial}(\boldsymbol{P}) \approx S_x[m(y-1)+1 : m(y-1)+m^2]$. The positions $m(y-1)+1$ satisfying the latter condition can be found in $O(nm+m^2 \log m)$ time by 1d-OPPM algorithms, which we showed in the previous sections or KMP-based ones [37, 35], because $|S_x| = nm$ and $|\mathit{serial}(\boldsymbol{P})| = m^2$. Because we need the preprocess for the pattern *serial*($\boldsymbol{P}$) only once, and execute the search in S_x for each x, the result follows. ■ ■

In the rest of this paper, we try a direct approach to two-dimensional strings based on the duel-and-sweep paradigm, inspired by the work [2, 19]. A substring of $\boldsymbol{T}$ of size $m \times m$ will be referred as a candidate. $\boldsymbol{T}_{x,y}$ denotes a candidate with the top-left corner at (x, y).

Pattern preprocessing

For $0 \le a < m$ and $-m < b < m$, we say that a pair $\langle (i_x, i_y), (j_x, j_y) \rangle$ of locations is *a witness pair for the offset* (a, b) if either of the following holds:

- $\boldsymbol{P}[i_x, i_y] = \boldsymbol{P}[j_x, j_y]$ and $\boldsymbol{P}[i_x + a, i_y + b] \neq \boldsymbol{P}[j_x, j_y]$,

- $\boldsymbol{P}[i_x, i_y] > \boldsymbol{P}[j_x, j_y]$ and $\boldsymbol{P}[i_x + a, i_y + b] \leq \boldsymbol{P}[j_x, j_y]$,

- $\boldsymbol{P}[i_x, i_y] < \boldsymbol{P}[j_x, j_y]$ and $\boldsymbol{P}[i_x + a, i_y + b] \geq \boldsymbol{P}[j_x, j_y]$.

The *witness table W for pattern $\boldsymbol{P}$* is a two-dimensional array of size $m \times (2m-1)$, where $W[a, b]$ is a witness pair for the offset (a, b). If the overlap regions are order-isomorphic when $\boldsymbol{P}$ is superimposed with offset (a, b), then no witness pair exists. We denote it as $W[a, b] = \langle (m+1, m+1), (m+1, m+1) \rangle$.

We show how to efficiently construct the witness table W. For $\boldsymbol{P}$ and each $0 \leq a < m$, we define the *Z-array* $Z_{\boldsymbol{P},a}$ by

$$Z_{\boldsymbol{P},a}[i] = \max_{1 \leq j \leq |P_1| - i + 1} \{ j \mid P_1[1 : j] \approx P_2[i : i + j - 1] \} \text{ for each } 1 \leq i \leq |P_1|,$$

where $P_1 = \mathit{serial}(\boldsymbol{P}[1 : m - a, 1 : m])$, $P_2 = \mathit{serial}(\boldsymbol{P}[a + 1 : m, 1 : m])$, and $|P_1| = |P_2| = m(m - a)$.

Lemma 3.30. *For arbitrarily fixed $a \geq 0$, we can compute the value of $W[a, b]$ in $O(1)$ time and for each b, assuming that $Z_{\boldsymbol{P},a}$ is already computed.*

Proof: For an offset (a, b) with $b \geq 0$, let us consider $z_{a,b} = Z_{\boldsymbol{P},a}[b \cdot (m - a) + 1]$.

Case 1 $z_{a,b} = (m - a) \cdot (m - b)$: Note that the value is equal to the number of elements in the overlap region. Then $\boldsymbol{P}[1 : m - a, 1 : m - b] \approx \boldsymbol{P}[a + 1 : m, b + 1 : m]$, so that no witness pair exists for the offset (a, b).

Case 2 $z_{a,b} < (m - a) \cdot (m - b)$: There exists a witness pair $\langle (i_x, i_y), (j_x, j_y) \rangle$, where (j_x, j_y) is the location of the element in $\boldsymbol{P}$, that corresponds to the $(z_{a,b} + 1)$-th element of $P_1 = \mathit{serial}(\boldsymbol{P}[1 : m - a, 1 : m])$. By a simple calculation, we can obtain the values (j_x, j_y) in $O(1)$ time. We can also compute (i_x, i_y) from (j_x, j_y) in $O(1)$ time, similarly to the proof of Lemma 3.3, with the help of auxiliary arrays $\mathit{Lmax}_{\boldsymbol{P},a}$ and $\mathit{Lmin}_{\boldsymbol{P},a}$. (Details are omitted.)

Symmetrically, we can compute it for $b < 0$. ■

36	47	20	9	49
42	44	31	8	11
17	39	28	12	23
22	12	16	15	27
24	29	11	42	49

Figure 3.7: An example of witness pair. The pattern $\boldsymbol{P}$ is shown on the left and the alignment of P with itself with offset $(3, 2)$ is shown on the right. The pair $\langle(2, 1), (2, 2)\rangle$ is a witness pair for offset $(3, 2)$, since $\boldsymbol{P}[2, 1] = 47 > 44 = \boldsymbol{P}[2, 2]$, but $\boldsymbol{P}[5, 3] = 23 < 27 = \boldsymbol{P}[5, 4]$.

Table 3.3: Computation of $Z_{\boldsymbol{P},3}$. For $\boldsymbol{P}$ in Fig. 3.7, the overlap regions for offset $(3, 0)$ are traversed in left-to-right/top-to-bottom order to obtain P_1 and P_2.

	1	2	3	4	5	6	7	8	9	10
P_1	36	47	42	44	17	39	22	12	24	29
P_2	9	49	8	11	12	23	15	27	42	49
$Z_{\boldsymbol{P},3}$	2	1	2	2	3	1	2	2	2	1

Lemma 3.31. *We can construct the witness table W in $O(m^3)$ time.*

Proof: Assume that we sorted all elements of $\boldsymbol{P}$. For an arbitrarily fixed a, calculation of $Lmax_{\boldsymbol{P},a}$ and $Lmin_{\boldsymbol{P},a}$ takes $O(m^2)$ time by using sorted $\boldsymbol{P}$. $Z_{\boldsymbol{P},a}$ can be constructed in $O(m^2)$ time by Lemma 3.22. Furthermore, finding witness pairs for all offsets (a, b) takes $O(m)$ time by Lemma 3.30. Since there are m such a's to consider, W can be constructed in $O(m^3)$ time. ■

Dueling stage

We can show the transitivity as follows.

Lemma 3.32. *For any $a, b, a', b' \geq 0$, let us consider three candidates $\boldsymbol{T}_1 = \boldsymbol{T}_{x,y}$, $\boldsymbol{T}_2 = \boldsymbol{T}_{x+a,y+b}$, and $\boldsymbol{T}_3 = \boldsymbol{T}_{x+a',y+b'}$. If $\boldsymbol{T}_1$ is consistent with $\boldsymbol{T}_2$ and $\boldsymbol{T}_2$ is consistent with $\boldsymbol{T}_3$, then $\boldsymbol{T}_1$ is consistent with $\boldsymbol{T}_3$.*

Lemma 3.33. *([2]) The dueling stage can be done in $O(n^2)$ time by using W.*

Table 3.4: Witness pairs for offsets $(3,0)$, $(3,1)$, $(3,2)$, $(3,3)$, $(3,3)$ for $\boldsymbol{P}$ in Fig. 3.7.

(a,b)	$(3,0)$	$(3,1)$	$(3,2)$	$(3,3)$	$(3,4)$
$z_{a,b}$	2	2	3	2	2
$W[a,b]$	$\langle(1,1),(2,1)\rangle$	$\langle(1,2),(2,1)\rangle$	$\langle(2,1),(2,2)\rangle$	$\langle(1,2),(2,1)\rangle$	$\langle(5,5),(5,5)\rangle$

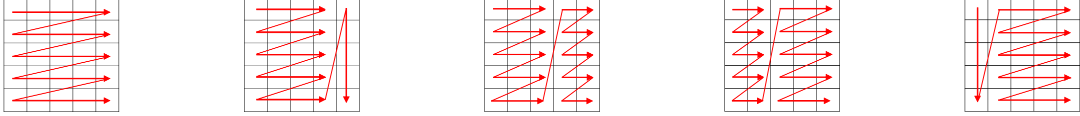


Figure 3.8: Example of traversing directions that we use for sweeping algorithm.

Sweeping stage

We first consider two surviving candidates $\boldsymbol{T}_{x,y_1}$ and $\boldsymbol{T}_{x,y_2}$ in some column x, with $y_1 < y_2$. If we traverse $\boldsymbol{T}[x:x+m-1, 1:n]$ from top-to-bottom/left-to-right manner we can reduce the problem to one-dimensional order-preserving problem. Thus performing the sweeping stage for some column x will take $O(nm)$ time. Since there are $n-m-1$ such columns, the sweeping stage will take $O(n^2m)$ time.

Next, we propose a method that takes advantage of consistency relation in both horizontal and vertical directions. First, we construct m strings $P_i = \mathit{serial}(\boldsymbol{P}[1:m-i, 1:m])\mathit{serial}(\boldsymbol{P}[m-i+1:m, 1:m])$ for $0 \leq i < m$ by serializing $\boldsymbol{P}$ in different way. We then compute Lmax_{P_i} and Lmin_{P_i} for $0 \leq i < m$, thus we can compare the order-isomorphism of the pattern with the text in several different ways. Lmax_{P_i} and Lmin_{P_i} for $0 \leq i < m$ can be computed in $O(n^3)$ time by sorting $\mathit{serial}(\boldsymbol{P})$ once and then calculated Lmax_{P_i} and Lmin_{P_i} by using the sorted $\mathit{serial}(\boldsymbol{P})$. Fig. 3.8 shows P_i for $0 \leq i < m$ where $m = 5$. We also do the same computation for bottom-to-top/left-to-right traversing direction.

Let us consider two overlapping candidates $\boldsymbol{T}_{x_1,y_1}$ and $\boldsymbol{T}_{x_2,y_2}$, where $x_1 < x_2$ and $y_1 < y_2$. Suppose that $\boldsymbol{T}_{x_1,y_1}$ is order-isomorphic to the pattern and we need to check $\boldsymbol{T}_{x_2,y_2}$. Since $\boldsymbol{T}_{x_1,y_1}$ is consistent with $\boldsymbol{T}_{x_2,y_2}$, we need to check the order-isomorphishm of the region of $\boldsymbol{T}_{x_2,y_2}$ that is not an overlap region. We do this by using P_j, where $j = x_2 - x_1$, without checking the overlap region. This idea is illustrated in Figure 3.9 (a). The procedure for $y_1 > y_2$ is symmetrical.

Next, consider three overlapping candidates $\boldsymbol{T}_1 = \boldsymbol{T}_{x_1,y_1}$, $\boldsymbol{T}_2 = \boldsymbol{T}_{x_2,y_2}$ and $\boldsymbol{T}_3 = \boldsymbol{T}_{x_3,y_3}$,

Figure 3.9: (a) Elements in the overlap region is checked only once. (b) Elements in the blue region must be checked twice.

such that $x_1 \leq x_2 \leq x_3$ and $y_2 \leq y_3$. We assume that $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ are both order-isomorphic to the pattern. If $y_1 \leq y_2$, we can use the method for two overlapping candidates that we described before to perform sweeping efficiently. However, if $y_1 \geq y_2$, as showed in Fig. 3.9 (b), we need to check the blue region twice since we do not know the order-isomorphism relation between the blue region with the overlap region of $\boldsymbol{T}_2$ and $\boldsymbol{T}_3$.

By using the above method, we can reduce the number of comparisons for sweep stage. However, the time complexity remains the same.

Lemma 3.34. *The sweeping stage can be completed in $O(n^2m)$ time.*

By Lemmas 3.31, 3.33, and 3.34, we conclude this section as follows.

Theorem 3.35. *The duel-and-sweep algorithm solves 2d-OPPM Problem in $O(n^2m+m^3)$ time.*

Chapter 4

Parallel duel-and-sweep algorithms for SCER

In this chapter, we show a parallel version of the duel-and-sweep algorithm for SCER. To efficiently solve the SCER problem in parallel, we enrich ideas used in the serial algorithm with new ones. To ensure the generality, our parallel algorithm uses $\approx$-encoding. In this chapter, $\widetilde{X}$ means an $\approx$-encoding of string X, unless otherwise stated. For a string X, we suppose that $\widetilde{X}$ can be computed in $\tau^{\mathrm{t}}_{|X|}$ time and $\tau^{\mathrm{w}}_{|X|}$ work on P-CRCW PRAM. Assuming that $\widetilde{X}$ has been computed, we suppose that $\widetilde{X}_k[i]$ can be computed in $\xi^{\mathrm{t}}_{|X[i:]|}$ time and $\xi^{\mathrm{w}}_{|X[i:]|}$ work on P-CRCW RPAM.

Hereinafter, in our pseudo-codes we will use "$\leftarrow$" to note assignment operation without conflict (only one processor writes into the memory location). In case of the assignment with possible conflicts (multiple processors attempt to access the same memory location), we will note it as "$\Leftarrow$". In case of the conflict, the processor with the smallest index succeeds in writing into the memory.

Lemma 4.1. *For strings X and Y of equal length n, given $\widetilde{X}$ and $\widetilde{Y}$, Algorithm 24 computes a tight mismatch position pair in $O(1)$ time and $O(n)$ work on the P-CRCW PRAM.*

Algorithm 8: Check in parallel whether X and Y match, given $\widetilde{X}$ and $\widetilde{Y}$

```
1 Function CheckParallel(X̃, Ỹ)
2     w ← 0;
3     for each i ∈ {1, ..., |X|} do in parallel
4         if X̃[i] ≠ Ỹ[i] then
5             w ⇐ i;
6     return w;
```

Proof: In Algorithm 8, for each element of X, we "attach" a processor at position of X. If $\widetilde{X}[i] \neq \widetilde{Y}[i]$ for some i, the corresponding processor tries to update the shared variable w. Recall that in P-CRCW PRAM, the processor with the lowest index will succeed in writing into w. Thus, at the end of the algorithm w contains the tight mismatch position.

4.1 Aperiodic pattern case for SCER

In this section, we consider the case, when the pattern is aperiodic. That is, the length of the smallest block-based period $p > 1$ is at least the half of the length of P. It is useful to consider a fast algorithm for an aperiodic pattern, because the probability for a random generated string being aperiodic is very close to 1. The algorithm for the aperiodic pattern case is simpler than that of the periodic case. The work complexity of the SCER algorithm in the aperiodic pattern case is less than that of the general case by a factor of $\log m$. The algorithm for the aperiodic case only computes the first half of a witness table. Actually, using only the first half of the witness table does not affect the computational complexity of the pattern searching algorithm. Since the pattern is aperiodic, for $i \in \{1, \ldots, \lfloor m/2 \rfloor\}$, $\mathcal{W}_P(i) \neq \emptyset$. After the construction of a witness table, the pattern searching algorithm performs $O(\log m)$ rounds of parallel duels, until there is at most one surviving candidate, for each $2^{\lfloor \log m \rfloor - 1}$-block of the text. Checking the surviving candidates naively gives an $O(\xi^{\mathrm{t}}_m \log m)$ time and $O(\xi^{\mathrm{w}}_m m)$ work pattern searching algorithm on P-CRCW PRAM.

Pattern preprocessing

Recall that the goal of the preprocessing stage is to compute the witness table W, where $W[a] = 0$ if $\mathcal{W}_P(a) = \emptyset$, and $W[a] \in \mathcal{W}_P(a)$ otherwise. Since the pattern is aperiodic, for $i \in \{0, \ldots, \lfloor m/2 \rfloor\}$, $\mathcal{W} \neq \emptyset$. Using this, the pattern preprocessing construct the first half of a witness table W, that is $W[0 : \lfloor m/2 \rfloor]$. It can be shown that using the first half of a witness table does not affect the computational complexity of the pattern searching. The preprocessing algorithm when the pattern is aperiodic is described in Algorithm 11. Initially, all entries of W are set to zero, and at any point of the algorithm execution all positions i of W satisfy the following,

$$W[i] \neq 0 \text{ implies } W[i] \in \mathcal{W}_P(i).$$

Following Vishkin's algorithm, we consider partitioning W into blocks of size 2^k. We will call each block a 2^k-*block*, with the last 2^k-block possibly being shorter than 2^k. That is, the 2^k-blocks are $W[i \cdot 2^k : (i+1) \cdot 2^k - 1]$ for $i = 0, \ldots, \lfloor m/2^k \rfloor - 1$ and $W[\lfloor m/2^k \rfloor \cdot 2^k : m]$. The value k will be incremented with each iteration of the while-loop in Algorithm 14. Using the value of k, we refer to each iteration as *the round* k. If for some $W[0 : i]$, each 2^k-block of $W[0 : i]$ contains one zero entry, we will say that $W[0 : i]$ satisfies the 2^k-*sparsity* property. (For the first 2^k-block, since $W[0] = 0$, $W[1 : 2^k - 1]$ does not contain any zeros.) At the beginning of round k, the following properties hold.

- W satisfies the 2^k-sparsity.
- For $1 \leq i \leq \lfloor m/2 \rfloor$, $W[i] \leq 2^{k+2}$.

Consider round k. For each round, the position p_k of the unique zero in the second 2^k-block becomes a *suspected period*. Since the pattern is aperiodic, $\mathcal{W}_P(p_k) \neq \emptyset$. Using the information in the first 2^k-block of W, the 2^k-sparsity property is satisfied by performing $(\lfloor m/2 \rfloor)/2^k$ parallel *duels* with respect to the pattern. The duel w.r.t. to the pattern is

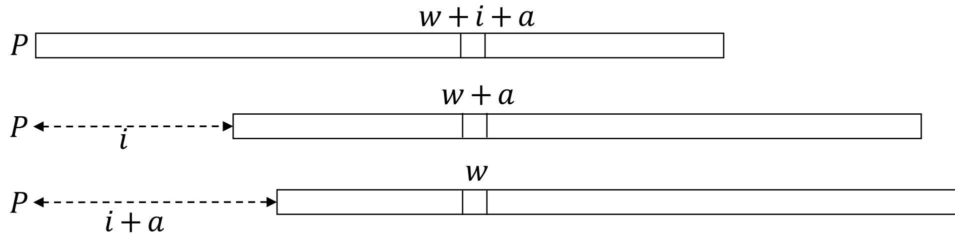


Figure 4.1: Dueling with respect to the pattern for offsets i and $i + a$.

same to the one described in the dueling stage of the serial algorithm, except that instead of superimposing two copies of the pattern on the text, we superimpose them on the pattern (Figure 4.1).

Lemma 4.2. *Suppose $w \in \mathcal{W}_P(a)$, $j \leq m - w$ and $j - i = a$. Then,*

1. *if the offset j survives the duel, i.e., $\widetilde{P}_{j+1}[w] = \widetilde{P}[w]$, then $w + a \in \mathcal{W}_P(i)$;*
2. *if the offset i survives the duel, i.e., $\widetilde{P}_{j+1}[w] \neq \widetilde{P}[w]$, then $w \in \mathcal{W}_P(j)$.*

Proof: If $\widetilde{P}_{j+1}[w] \neq \widetilde{P}_1[w]$, then $w \in \mathcal{W}_P(j)$ by definition. Suppose $\widetilde{P}_{j+1}[w] = \widetilde{P}_1[w]$. The fact $w \in \mathcal{W}_P(a)$ means $\widetilde{P}_1[w] \neq \widetilde{P}_{a+1}[w]$ and thus $\widetilde{P}_{j+1}[w] \neq \widetilde{P}_{a+1}[w]$. By Property (3) of the $\approx$-encoding (Definition 2.3), we have $\widetilde{P}_{i+1}[w + a] \neq \widetilde{P}_1[w + a]$, which means $w + a \in \mathcal{W}_P(i)$. ■

At this point, $W[p_k] \neq 0$. Recall that $2_k \leq p_k < 2^{k+1}$. Now, let us consider any 2^{k+1}-block B that is not the first 2^{k+1}-block. Since B satisfies the 2^k-sparsity, there are at most two zeros in B. If B has less than two zero entries, B already satisfies the 2^k-sparsity. Otherwise, let i and $i+a$ be their positions. Since $a < 2^{k+1}$, $W[a] \neq 0$. Using the dueling w.r.t. the pattern, we can update one of $W[i]$ and $W[i + a]$. Thus, after this procedure to all 2^{k+1}-blocks, W satisfies the 2^k-sparsity. The procedure for satisfying the 2^{k+1}-sparsity is described in Algorithm 10. We prepare a technical function `GetZeros`(l, r, k) in Algorithm 9, which returns positions $i \in \{l, \dots, r\}$ such that $W[i] = 0$, assuming that $W[0 : r]$ satisfies the 2^k-sparsity. Algorithm 9 runs in $O(1)$ time and $O(r)$ work on the P-CRCW PRAM.

Algorithm 9: Assuming that $W[0:r]$ is 2^k-sparse, returns positions of zeros in $W[l:r]$.

```
1 Function GetZeros(l, r, k)
2     create array A[0 : ⌊r/2^k⌋ − ⌊l/2^k⌋] and initialize elements to −1;
3     for each i ∈ {l, ..., r} do in parallel
4         if W[i] = 0 then A[⌊i/2^k⌋ − ⌊l/2^k⌋] ⇐ i;
5     return A;
```

Algorithm 10: Satisfy 2^{k+1}-sparsity of $W[0:x]$.

```
1 Function SatisfySparsity(x, k)
2     A ← GetZeros(2^{k+1}, x, k);
3     for each i ∈ {0, 1, ..., ⌊|A|/2 − 1⌋} do in parallel
4         j1 ← A[2i], j2 ← A[2i + 1];
5         if j1 ≠ −1 and j2 ≠ −1 then
6             surv ← Dueling(P̃, j1, j2);
7             a ← j2 − j1;
8             if surv = j1 then
9                 W[j2] ⇐ W[a];
10            if surv = j2 then
11                W[j1] ⇐ W[a] + a;
```

Now, to ensure the second invariant property, we take advantage of the following block-based periodicity properties. Suppose that p is the smallest block-based period of some prefix of P, say $P[1:x]$.

1. For offset $i \in \{1, \ldots, x\}$ such that $i \equiv 0 \pmod p$, there are no witnesses. This fact can be derived from Lemma 2.7.

2. For offset $i \in \{1, \ldots, x-p\}$ such that $i \not\equiv 0 \pmod p$, $\mathcal{W}_P(i) \neq \emptyset$. This fact can be derived using the well-known periodicity lemma for exact matching period [26]. The periodicity lemma states that if a string X has periods p and q such that $p+q \leq m$, then $gcd(p,q)$ is also a period of X.

3. For offset $i \in \{1, \ldots, x-p\}$ such that $i \not\equiv 0 \pmod p$, if w is a witness for offset $(i \bmod p)$ such that $w \leq p$, then w is also a witness for offset i. For offset $i \in \{1, \ldots, p-1\}$, there always exists a witness w such that $w \leq p$.

Algorithm 11: Algorithm for the pattern preprocessing (aperiodic pattern)

```
1  Function PreprocessingParallelAperiodic(P̃)
2      Initialize W[0 : ⌊m/2⌋] to 0;
3      k ← 0;
4      while 2^{k+2} ≤ m do
5          p ← GetZeros(2^k, 2^{k+1} − 1, k)[0];
6          c ← CertaintySatisfiedUntil(p, k);
7          for k' = k + 1 to c − 1 do
8              SatisfySparsity(m', k');
9          k ← c − 1;
10     Z ← GetZeros(0, ⌊m/2⌋, k);
11     for i = 0 to |Z| − 1 do
12         z ← Z[i];
13         if z ≠ −1 then
14             W[z] ⇐ CheckParallel(P̃[1 : m − z], P̃_{z+1}[1 : m − z]);
```

Suppose that p_k is a block-based period of $P[1 : 2^c]$, but not of $P[1 : 2^{c+1}]$ for some $c > k + 1$. Let w be the tight witness for offset p_k. By the block-based periodicity, we have the following. Since p_k is the smallest non-zero block-based period of $P[1 : 2^c]$, we can used the observations in the previous paragraph to fill in witness for $W[0 : 2^c - 1]$. The procedure is described in Algorithm 12.

For the sake of convenience, we absorb the case when $c = k + 1$ into Algorithm 12. If $c = k + 1$, Algorithm 12 only updates $W[p_k]$. If $c > k + 1$, Algorithm 12 copies the contents of $W[0 : p_k - 1]$ periodically into $W[0 : 2^c - p_k]$. That is, for each $i \leq 2^c - p$ such that $i \bmod p_k \neq 0$, the value of $W[i \bmod p_k]$ is copied into $W[i]$. For $i \leq 2^c - p$ such that $i \bmod p_k = 0$, $(W[p_k] - (i - p)) \in \mathcal{W}_P(i)$. This is illustrated in Figure 4.2.

Lemma 4.3. *Suppose c is an integer such that p_k is a block-based period of $P[1 : 2^c]$, but not of $P[1 : 2^{c+1}]$. Let w be a witness for offset p_k, such that $2^c < w \leq 2^{c+1}$. For*

$i \in \{p_k, \ldots, 2^c - 2^{\lfloor \log p_k \rfloor + 2}\}$,

$$\mathcal{W}_P(i) \ni \begin{cases} W[p_k] - (i - p_k), & \textit{if } i \bmod p_k = 0, \\ W[i \bmod p_k], & \textit{otherwise.} \end{cases}$$

Proof: Let $i_w = \max i \mid 1 \leq i_w < w$ and $i_w \bmod p = 0$. Since p_k is a block-based period of $P[1{:}2^c]$, but not of $P[1{:}2^{c+1}]$, $P[i_w{+}1{:}i_w{+}p_k] \not\approx P[1{:}p_k]$. Thus, for $i \in \{p_k, \ldots, i_w{-}1\}$ such that $i \bmod p_k = 0$, $w - (i - p_k)$ is a mismatch position for $P[i + 1w + p_k] \not\approx P[1 : w - (i - p_k)]$. Since $i_w > 2^c - p_k$ and $2^{\lfloor \log p_k \rfloor} \leq p_k < 2^{\lfloor \log p_k \rfloor + 1}$, for $i \in \{p_k, \ldots, 2^c - 2^{\lfloor \log p_k \rfloor + 2}\}$ $w - (i - p_k) \in \mathcal{W}_P(i)$.

Now, let us consider $j \in \{p_k, \ldots, 2^c - 2^{\lfloor \log p_k \rfloor + 2}\}$ such that $i \bmod p \neq 0$. Let $k = \lfloor \log m \rfloor + 1$. By the algorithm invariant, for positions $i \neq 0$ of the first 2^k-block, $W[i] \leq 2^{k+1}$. Let $i = \max i \mid i < j$ and $i \bmod p = 0$. Thus, duels between offsets i and j using witness $W[i]$ are in range with respect to $P[1 : 2^c]$. By the discussions in the previous paragraph, $P[i + 1 : i + p_k] \approx P[1 : p_k]$. Since $w + p_k > 2^c$, offset i will always win the duel. Thus, $W[j \bmod p] \in \mathcal{W}_P(j)$. ∎

For $i \in \{2^c - p + 1, \ldots, 2^c - 1\}$, the algorithm naively verifies every zero position, whether i is a block-based period of $P[1 : 2^{c+1}]$. At this points $W[0 : 2^c - 1]$ satisfies the 2^k-sparsity and $2^k \leq p_k \leq 2^{k+1}$. Thus, there are at most three zero positions in $W[2^c - p + 1 : 2^c - 1]$. Furthermore, by the periodicity lemma, at most one of such i is a block-based period of $P[1{:}2^{c+1}]$. After the termination of Algorithm 12, $W[0 : 2^c - 1]$ satisfies the 2^{c-1}-sparsity and, for $i \in \{0, \ldots, 2^c - 1\}$, $W[i] \in \mathcal{W}_P(i)$. At this point, the control goes back to Algorithm 11. Algorithm 11 incrementally satisfies $2^{k'}$-sparsity for $k + 1 \leq k' \leq c$ and the next round starts at $k = c - 1$. We prove the second invariant of Algorithm 11.

Lemma 4.4. *At the beginning of round k, for all $i \in \{0, \ldots, \lfloor m/2 \rfloor\}$, it holds $W[i] \leq 2^{k+2}$.*

Proof: We show the lemma by induction on k. At the beginning of round 0, every

Algorithm 12: Simultaneously satisfies certainty property for across multiple rounds

```
1  Function CertaintySatisfiedUntil(p, k)
2      c ← k + 1;
3      while 2^c ≤ ⌊m/2⌋ and W[p] = 0 do
4          for each i ∈ {2^c + 1, ..., 2^(c+1)} do in parallel
5              if P̃_(p+1)[i − p] ≠ P̃[i mod p + 1] then
6                  W[p] ⇐ i − p;
7          c ← c + 1;
8      c ← c − 1;
9      for each i ∈ {p, ..., 2^c − p} s.t. W[i] = 0 do in parallel
10         if i mod p = 0 then
11             W[i] ⇐ W[p] − (i − p);
12         else
13             W[i] ⇐ W[i mod p];
14     Z ← GetZeros(2^c − p + 1, 2^c − 1, k);
15     for i = 0 to |Z| − 1 do
16         z ← Z[i];
17         if z ≠ −1 then
18             W[z] ⇐ CheckParallel(P̃[1 : 2^(c+1) − z], P̃_(z+1)[1 : 2^(c+1) − z]);
19     return c;
```

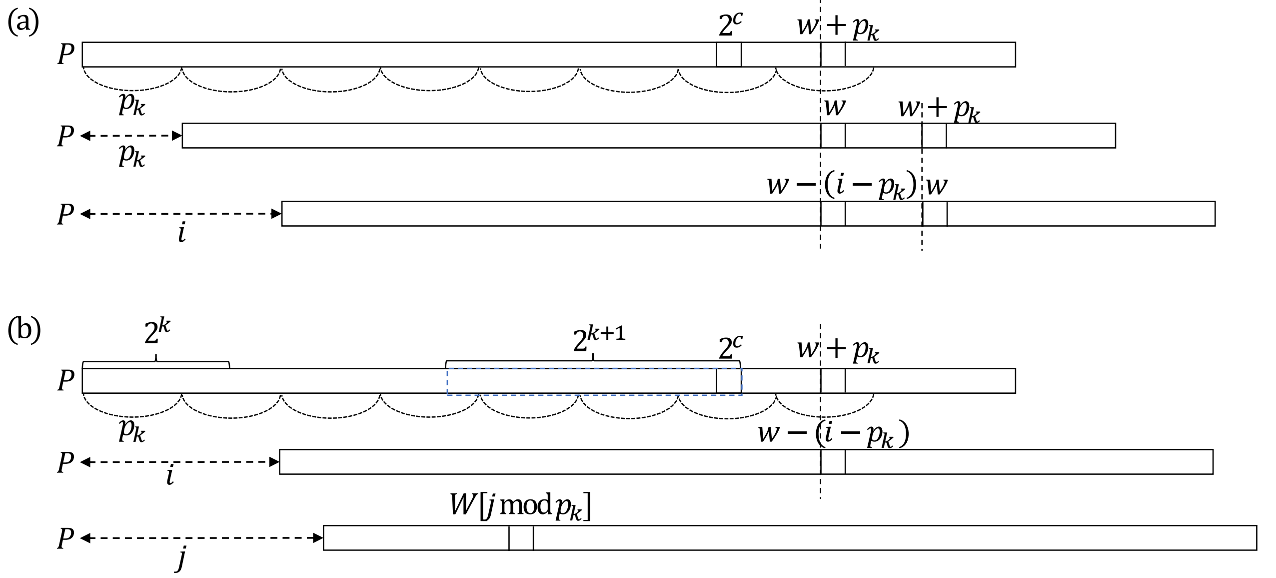


Figure 4.2: Illustration to Lemma 4.3. w is a witness for offset p_k. (a) Case when $i \bmod p = 0$. For $i \in \{p_k, \ldots, 2^c - 2^{\lfloor \log p \rfloor + 2}\}$, $w - (i - p_k) \in \mathcal{W}_P(i)$. (b) Case when $i \bmod p = 0$. For $i \in \{p_k, \ldots, 2^c - 2^{\lfloor \log p \rfloor + 2}\}$, $W[i \bmod p_k] \in \mathcal{W}_P(i)$.

element of W is zero, thus, the claim holds. We will show that the lemma holds for k assuming that it is the case for $k-1$.

For round k, suppose that p_k is a block-based period of $P[1:2^c]$, but not of $P[1:2^{c+1}]$ where $c \geq k+1$. For $i \in \{0, \ldots, 2^c - p_k\}$, $W[i] \leq 2^{c+1}$. For $i \in \{2^c - p_k + 1, \ldots, 2^c - 1\}$ such that $W[i] = 0$, the algorithm naively checks if i is a block-based period of $P[1:2^{c+1}]$ or not. Thus, for $i \in \{2^c - p_k + 1, \ldots, 2^c - 1\}$, $W[i] \leq 2^{c+1}$.

Now let us consider duels between offsets i and j such that $0 < j - i < 2^{k+1}$. Let $a = j - i$. Suppose $a \bmod p_k \neq 0$. If $W[a]$ was updated during the previous rounds, then by the algorithm invariant, $W[a] \leq 2^k$. If $W[a]$ is updated during round k, then $W[a] = W[a \bmod p_k] \leq 2^k$. If i wins the duel, then $W[j] = W[a] \leq 2^k$. If j wins the duel, then $W[i] = W[a] + a \leq 2^k + 2^{k+1}$.

If $a \bmod p_k = 0$, $W[a] = W[p_k] - (a - p_k) < 2^{c+1}$. If i wins the duel, then $W[j] = W[a] \leq 2^{c+1}$. If j wins the duel, then $W[i] = W[a] + a = W[p_k] + p_k$. Since $W[p_k] = 2^c + x - p_k$ for some $0 < x \leq 2^c$, $W[p_k] + p_k = 2^c + x \leq 2^{c+1}$. Since the next starts at $k = c - 1$, we have shown that before round $k = c - 1$ for all i, $W[i] \leq 2^{k+2}$. ■

By the invariant property, before the round k, for all positions $0 \le i \le \lfloor m/2 \rfloor$, $W[i] \le 2^{k+2}$. When $2^{k+2} > m$, the while loop halts. At this point, the pattern W satisfies the 2^k-sparsity. Thus, there are at most four zeros left in the witness table. The algorithm checks them naively.

Theorem 4.5. *Given $\widetilde{P}$, a witness table can be computed in $O(\xi^{\mathrm{t}}_m \cdot \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m)$ work on the P-CRCW PRAM when the pattern is aperiodic.*

Proof: Let us consider round k. Satisfying 2^k-sparsity takes $m/2^k$ processors and $O(\xi(m))$ time. Finding the tight witness w for offset p_k takes $O(\xi^{\mathrm{t}}_m)$ time and $O(\xi^{\mathrm{w}}_m \cdot 2^{c+1})$ work, where $2^c < w \le 2^{c+1}$. Then, 2^{c-1}-sparsity is satisfied incrementally, which increases k to c. Since there are $O(\log m)$ rounds and $\sum_{k=0}^{\log m} 2^k = O(m)$, $\sum_{k=0}^{\log m} m/2^k = O(m)$, the total complexity is $O(\xi^{\mathrm{t}}_m \cdot \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m)$ work. ∎

Text processing

Now, we describe our parallel algorithm for the text processing when the pattern is aperiodic. We define array C of length $m+1$ and initialize every entry of C to *True*. (Recall that $n = 2m - 1$.) The pattern preprocessing updates C until $C[i] = \mathit{True}$ iff there is a pattern occurrence at i, that is $T_i \approx P$. We define the k-sparsity property for C as follows. Each k-block of C contains at most one location i such that $C[i] = \mathit{True}$.

Since the pattern is aperiodic, a witness table does not contain zero positions, except $W[0]$. After $\lfloor \log m \rfloor - 1$ rounds of parallel duels, C satisfies the $(\lfloor \log m \rfloor - 1)$-sparsity property. Hence, there are at most four positions in C, whose value is *True*. The algorithm checks these locations naively for an occurrence. The pattern searching is described in Algorithm 13.

Theorem 4.6. *Given a witness table, $\widetilde{P}$, and $\widetilde{T}$, the pattern searching solves the pattern searching problem under SCER in $O(\xi^{\mathrm{t}}_m \cdot \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot n)$ work on the P-CRCW PRAM.*

Algorithm 13: Pattern searching (aperiodic pattern)

```
1  Initialize array C of length m to True;
2  Round k ≤ 0;
3  while 2^k < ⌊m/2⌋ do
4      for each 2^k-block B of C do in parallel
5          Let j_1 be the position of zero the first half of B;
6          Let j_2 be the position of zero the second half of B;
7          a ← j_2 − j_1;
8          if W[a] ≠ 0 and a < ⌊m/2⌋ then
9              surv ← Duel(T̃_{j_2+1}, j_1, j_2);
10             if surv ≠ j_1 then
11                 C[j_1] = False;
12             else if surv ≠ j_2 then
13                 C[j_2] = False;
14     k ← k + 1;
15 for each i ∈ {1, ..., m} do
16     if C[i] = True then
17         w ⇐ CheckParallel(T̃_i, P̃);
18         if w ≠ 0 then
19             C[i] ← False;
```

4.2 General case for SCER

4.2.1 Pattern preprocessing

Let us consider how to construct the witness table for the exact matching problem. Suppose that P is periodic, that is, the smallest period $p > 0$ of P satisfies $2p \leq m$. The algorithm is based on the following facts, that can be derived using the well-known periodicity lemma, which states that if a string X has periods p and q such that $p + q \leq m$, then $gcd(p, q)$ is also a period of X.

1. For offset $i \in \{1, \ldots, \lceil m/2 \rceil\}$ such that $i \equiv 0 \pmod{p}$, there are no witnesses.

2. For offset $i \in \{1, \ldots, \lceil m/2 \rceil\}$ such that $i \not\equiv 0 \pmod{p}$, if w is a witness for offset $(i \bmod p)$, then w is also a witness for offset i.

Vishkin's algorithm [44] finds the smallest period p, while locating witnesses for offsets less than p. Once the smallest period p is found and $W[i]$ is updated for each $1 \leq i < p$, the rest of the work is simply to copy the contents for $W[i]$ periodically for each $i > p$. On the other hand, if P does not have any period, the problem is even simpler; in the above process to find the smallest period p, all $W[i]$ for $1 \leq i < \lceil m/2 \rceil$ are updated to be non-zero witnesses, and we are done.

The periodicity lemma generally does not hold for a SCER [39]. This might result in a situation, where for an offset $i \in \{1, \ldots, \lceil m/2 \rceil\}$, satisfying $i \not\equiv 0 \pmod{p}$ does not imply that $\mathcal{W}_P(i) \neq \emptyset$.

Lemma 4.7. *The distance between two neighboring zeros decreases monotonically towards the end of* W.

Proof: Let a, b, c, d be positions of zeros in the witness table such that $a < b < c < d$, a neighbors b, and c neighbors d. For the sake of contradiction, suppose that $b - a < d - c$. Since both a and b are periods of P, $(b - a)$ is a period of $P' = P[1 : m - a]$. By

Lemma 2.7, all multiples of $(b-a)$ are also periods of P' and, by this extension, of P. Thus, $d-c \leq b-a$, leading to a contradiction. ■

Our preprocessing algorithm is described in Algorithm 14. To describe our algorithm rigorously, we define the following properties for a position i in the witness table.

- *witness certainty property (WCP) for position i:* $W[i] \neq 0$ implies $W[i] \in \mathcal{W}_P(i)$.
- *zero certainty property (ZCP) for position i:* $W[i] = 0$ implies $\mathcal{W}_P(i) = \emptyset$.

We say that position i is *finalized* if i satisfies the ZCP and the WCP. If for some l, r, all positions $i \in \{l, \ldots, r\}$ are finalized, then we say that $W[l : r]$ is *finalized.* Initially, all entries of the witness table are set to zero. During the entire preprocessing algorithm, each element of W is updated at most once. At any point of the execution of the preprocessing algorithm, all positions in W satisfy the WCP. The goal of the preprocessing algorithm is to update W in such way that all positions satisfy the ZCP.

Until the algorithm finds the smallest period $p_{\min}$, it selects the smallest zero position $p_k > 0$. We will show that at the beginning of the round k, the second 2^k-block contains an unique zero position and that unique zero position is p_k. Then, the witness table table is partitioned into *head* and *tail*, the lengths of which depend on the value of $LCP(p_k)$. To increase the efficiency, the algorithm tries to decrease the number of possible suspected periods by filling in as many witnesses as possible in the head. For the tail, the algorithm determines the positions of all zeros, while finding witnesses for the non-zero positions.

Initially the entire table is the head and the size of the tail is zero: $Head_0 = W$ and $Tail_0 = \varepsilon$. The head is shrunk and the tail is extended by the following rule. Let the *suspected period* p_k at round k be the first zero position after the index 0, i.e., p_k is the unique position in the second 2^k-block such that $W[p_k] = 0$. Then, $Head_{k+1} = W[0 : m-x-1]$ and $Tail_{k+1} = W[m-x : m-1]$ for $x = |Tail_{k+1}| = \max(|Tail_k| + 2^k,\ LCP_P(p_k))$. This is illustrated in Figure 4.3. When $|Head_k| < 2^k$, the 2^k-sparsity means that all the positions in the witness table are finalized. So, Algorithm 14 exits the while loop and

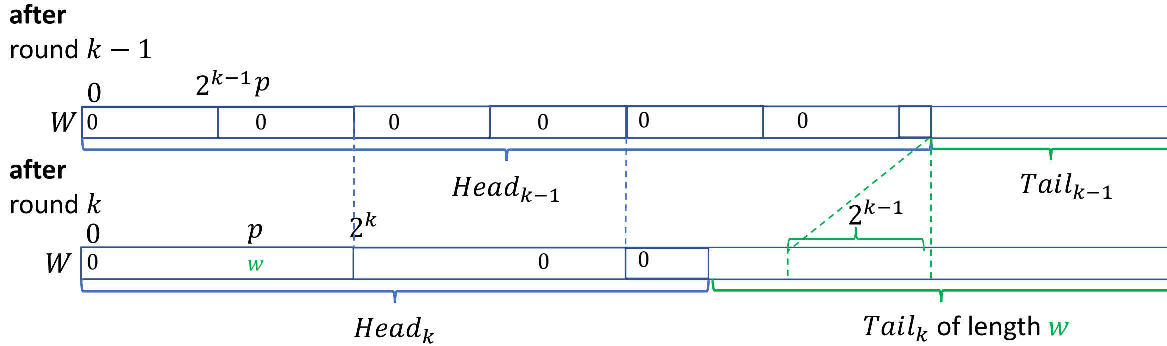


Figure 4.3: Illustration of the preprocessing invariant. W is partitioned into to parts, the duelable head and the non-duelable tail. $Head_{k-1}$ and $Head_k$ satisfies the 2^{k-1}-sparsity and 2^k-sparsity properties, respectively, while $Tail_{k-1}$ and $Tail_k$ satisfy the WCP and the ZCP. The length of $Tail_k$ is determined by the value of $LCP(p_k)$, where p_k is the suspected period for the round k. However, $Tail_k$ should longer than $Tail_{k-1}$ by at least 2^{k-1}.

halts. Formally, Algorithm 14 maintains W so that it satisfies the following invariant properties. At the beginning of round k,

- $Head_k$ is 2^k-sparse.
- For all positions i of $Head_k$ satisfy the ZCP.
- $Tail_k$ is finalized.

First, let us consider how the algorithm satisfies the 2^k-sparsity of $Head_k$. The procedure is described in Algorithm 10. At the beginning of round k, $Head_k$ already satisfies the 2^k-sparsity. The first 2^k-block of W is zero-free, except for $W[0]$, and the second 2^k-block contains an unique zero location p_k. In the case where the suspected period p_k is the smallest period of P, i.e., $\mathcal{W}_P(p_k) = \emptyset$, we have $tail = m - LCP_P(p_k) = p_k < 2^{k+1}$ when Algorithm 14 calls `SatisfySparsity(`$tail-1, k$`)`. Then the array A obtained at Line 2 is empty and `SatisfySparsity(`$tail - 1, k$`)` does nothing. After finalizing $Tail_{k+1}$, the algorithm will halt without going into the next loop, since $|Head_{k+1}| \leq m - LCP_P(p_k) = p_k < 2^{k+1}$. At that moment all positions of W are finalized.

Suppose that p_k is not a period of P. Let w be the tight mismatch position for offset p_k. After the algorithm sets $W[p_k] = w$, the first 2^{k+1}-block becomes zero-free, except for

Algorithm 14: Parallel algorithm for the pattern preprocessing

```
1  Function PreprocessingParallel()
2      tail ← m, k ← 0;                /* tail is the starting position of Tail_k */
3      while 2^k ≤ tail do
4          p ← GetZeros(2^k, 2^{k+1} − 1, k)[0];
5          W[p] ← CheckParallel(P̃[1 : m − p], P̃_{p+1}[1 : m − p]);
6          if W[p] = 0 then lcp ← m − p;
7          else lcp ← W[p] − 1;
8          old_tail ← tail;
9          tail ← min(old_tail − 2^k, m − lcp);
10         SatisfySparsity(tail − 1, k);
11         FinalizeTail(tail, old_tail, p, k);
12         k ← k + 1;
```

$W[0]$. We can update $Head_k$ so that it will satisfy the 2^k-sparsity by performing a duel between two offsets within the same 2^k-block.

Lemma 4.8. *For round k, suppose the preprocessing invariant holds true and $\mathcal{W}_P(p_k) \neq \emptyset$. Then, when* `SatisfySparsity` *is about to be called at Line 10 of Algorithm 14, for any two positions i, j of $Head_{k+1}$ such that $0 < j - i < 2^{k+1}$, $j \leq m - W[j - i]$.*

Proof: Let $a = j - i$ and $w = W[a]$. Recall that a belongs to the first 2^{k+1}-block and $W[a]$ is updated only if $a = p_k$. Suppose $a \neq p_k$. At the beginning of round k, by the invariant property, we have $w \leq |Tail_k| + 2^k$. Since $j < |Head_{k+1}| = m - |Tail_{k+1}|$, $j + w \leq j + |Tail_k| + 2^k < m - |Tail_{k+1}| + |Tail_k| + 2^k$. Since $|Tail_{k+1}| - |Tail_k| \geq 2^k$, $m - |Tail_{k+1}| + |Tail_k| + 2^k < m$. Thus, $j + w \leq m$.

If $a = p_k$, $w = W[p_k]$ is the tight witness for offset p_k, i.e., $w = LCP_P(p_k) + 1$. Since $|Tail_{k+1}| \geq LCP_P(p_k)$, $j + w \leq j + |Tail_{k+1}| + 1$. Since $j < |Head_{k+1}|$, $j + |Tail_{k+1}| + 1 \leq |Head_{k+1}| + |Tail_{k+1}| \leq m$. We have proved that $j + w \leq m$. ■

Taking into account the fact that $Head_k$ is 2^k-sparse, the 2^{k+1}-sparsity of $Head_k$ can be satisfied in $O(1)$ time on $O(m/2^{k+1})$ processors. Since $Head_k$ is 2^k-sparse, every 2^{k+1}-block of $Head_k$ contains at most two zeros. If we perform a duel between the zero positions, one of them will be updated. The procedure is described in Algorithm 10.

Next, we prove the second invariant property.

Lemma 4.9. *At the beginning of round k, for all $i \in \{0, \dots, 2^k - 1\}$, it holds $W[i] \leq |Tail_k| + 1$ and for all $i \in \{2^k, \dots, |Head_k| - 1\}$, it holds $W[i] \leq |Tail_k| + 2^k$.*

Proof: We show the lemma by induction on k. At the beginning of round 0, every element of W is zero and $|Tail_0| = 0$, thus, the claim holds. We will show that the lemma holds for $k+1$ assuming that it is the case for k.

Suppose $i < 2^{k+1}$ and $i \neq p_k$. Then $W[i]$ is not updated. By induction hypothesis, $W[i] \leq |Tail_k| + 2^k \leq |Tail_{k+1}|$ holds. Suppose $i = p_k$. If $\mathcal{W}_P(p_k) = \emptyset$, the algorithm sets $W[p_k] = 0$ and thus the claim holds. If $\mathcal{W}_P(p_k) \neq \emptyset$, the algorithm sets $W[p_k]$ to the tight witness $LCP_P(p_k) + 1$. Thus, $W[p_k] = LCP_P(p_k) + 1 \leq |Tail_{k+1}| + 1$.

Suppose $2^{k+1} \leq i < |Head_{k+1}|$. If Algorithm 10 does not update $W[i]$, by the induction hypothesis, $W[i] \leq |Tail_k| + 2^k < |Tail_{k+1}| + 2^{k+1}$ holds. Suppose Algorithm 10 updates $W[i]$ or $W[j]$ by a duel between i and j, where $2^{k+1} \leq i < j < |Head_{k+1}|$ and $a = j - i < 2^{k+1}$. We have shown above that $W[a] \leq |Tail_{k+1}| + 1$. If i wins the duel, then $W[j] = W[a] \leq |Tail_{k+1}| + 1 \leq |Tail_{k+1}| + 2^{k+1}$. If j wins the duel, then $W[i] = W[a] + a \leq |Tail_{k+1}| + 1 + a \leq |Tail_{k+1}| + 2^{k+1}$. ■

Lemma 4.10. *headcomplexity In the round k of the while loop, Algorithm 10 updates the witness table so that $Head_{k+1}$ is 2^{k+1}-sparse in $O(\xi^{\mathrm{t}}_m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m/2^k)$ work on P-CRCW PRAM.*

Proof: Before the execution of Algorithm 10, since the preprocessing invariant is satisfied and $W[p_k]$ holds the tight witness for offset p_k, $W[0 : 2^k - 1]$ contains one zero. Now, let us consider a 2^k-block B of $Head_k$ that is not the first 2^k-block. Since $Head_k$ satisfies the 2^{k-1}-sparsity, there are at most two zero positions in B. Suppose that B has two distinct zero positions i and $i + a$. Since $a < 2^k$, $W[a] \neq 0$. By Lemma 4.8, for offsets i and $i + a$, $i + a \leq m - W[a]$. Thus, by Lemma 4.2, at least one of $W[i]$ and $W[i + a]$ is updated

as the result of the duel. Thus, after performing duels for all 2^k-blocks of $Head_k$, $Head_k$ satisfies the 2^k-sparsity.

Since each duel takes $O(\xi^{\mathrm{t}}_m)$ time and $O(\xi^{\mathrm{w}}_m)$ work and there are $O(m/2^k)$ duels in total, the overall time and work complexities are $O(\xi^{\mathrm{t}}_m)$ and $O(\xi^{\mathrm{w}}_m \cdot m/2^k)$, respectively. ■

Next, we discuss how Algorithm 15 finalizes $Tail_{k+1}$ for the round k. For the sake of convenience, we denote by $\mathcal{T}_k$ the set of positions of $Tail_k$. Since $Tail_k$ has already been finalized, it is enough to update $W[i]$ for $i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k$. Let us consider the case $|Tail_{k+1}| = |Tail_k| + 2^k$. Since by the invariant $Head_k$ satisfies the 2^k-sparsity, there are at most two zero positions in $\mathcal{T}_{k+1} \setminus \mathcal{T}_k$. It suffices to check the zero positions naively.

Now, let us consider the case when $|Tail_{k+1}| = LCP(p_k) > |Tail_k| + 2^k$.

Lemma 4.11. *Suppose $m - LCP_P(p) \le b < m$. If $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$ for any offset a such that $0 \le a \le b$ and $a \equiv b \pmod{p}$.*

Proof: Figure 4.7 may help understanding the proof. Suppose $w \in \mathcal{W}_P(b)$, i.e., $\widetilde{P}_{b+1}[w] \neq \widetilde{P}[w]$. Since p is a period of $P[1 : LCP_P(p)]$ and $a \equiv b \pmod{p}$, by Lemma 2.7, $(b - a)$ is also a period of $P[1 : LCP_P(p)]$, i.e., $P[1 + b - a : LCP_P(p)] \approx P[1 : LCP_P(p) - (b - a)]$. Particularly for the position $w \le m - b \le m - a$, we have $\widetilde{P}_{b-a+1}[w] = \widetilde{P}[w]$. Then, $\widetilde{P}_{b-a+1}[w] \neq \widetilde{P}_{b+1}[w]$ by the assumption (Figure 4.7). By Property (3) of the $\approx$-encoding (Definition 2.3), $\widetilde{P}_1[b - a + w] \neq \widetilde{P}_{a+1}[b - a + w]$. That is, $(w + b - a) \in \mathcal{W}_P(a)$. ■ Let us partition $\mathcal{T}_{k+1} \setminus \mathcal{T}_k$ into p_k subsets $\mathcal{S}_0, \dots, \mathcal{S}_{p_k-1}$ where $\mathcal{S}_{rem} = \{\, i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k \mid i \equiv rem \pmod{p_k} \,\}$, some of which can be empty. Lemma 4.11 implies that for each $rem \in \{0, \dots, p_k - 1\}$, there exists a boundary offset b_{rem} such that, for every $i \in \mathcal{S}_{rem}$, $\mathcal{W}_P(i) = \emptyset$ iff $i > b_{rem}$, unless $\mathcal{W}_P(i) = \emptyset$. Fortunately for many rem, one can find the boundary b_{rem} very easily, unless $\mathcal{S}_{rem} = \emptyset$. Let $q_{rem} = \max(\mathcal{S}_{rem})$ for non-empty $\mathcal{S}_{rem}$. Due to the 2^k-sparsity and the fact $p_k < 2^{k+1}$, it holds $W[q_{rem}] \neq 0$ for all but at most three rem. If $W[q_{rem}] \neq 0$, then q_{rem} is the boundary. By Lemma 4.11, $W[W[q_{rem}] + q_{rem} - i] \in \mathcal{W}_P(i)$ for all $i \in \mathcal{S}_{rem}$. Accordingly, Algorithm 15 updates those values $W[i]$ in parallel in Lines 9–11.

Algorithm 15: Finalize $Tail_{k+1}$.

```
1  Function FinalizeTail(tail, old_tail, p, k)
2      if old_tail − tail = 2^k then
3          Z ← GetZeros(tail, old_tail − 1, k);                    /* |Z| ≤ 2 */
4          for i = 0 to |Z| − 1 do
5              z ← Z[i];
6              if z ≠ −1 then
7                  W[z] ← CheckParallel(P̃[1 : m − z], P̃_{z+1}[1 : m − z])
8      else
9          for each i ∈ {tail, ..., old_tail − 1} do in parallel
10             q ← j where j ∈ {old_tail − p, ..., old_tail − 1} and j ≡ i (mod p);
11             if W[i] = 0 and W[q] ≠ 0 then W[i] ⇐ W[q] + q − i;
12         Z ← GetZeros(old_tail − p, old_tail − 1, k);            /* |Z| ≤ 3 */
13         for i = 0 to |Z| − 1 do
14             z ← Z[i];
15             if z ≠ −1 then Finalize(tail, old_tail, p, z mod p);
```

On the other hand, for rem such that $W[q_{rem}] = 0$, Algorithm 16 uses binary search to find b_{rem} and a witness $w \in \mathcal{W}_P(b_{rem})$ if it exists. Then, following Lemma 4.11, Algorithm 16 sets in parallel $W[i]$ to $w + (b_{rem} - i)$ where $w \in \mathcal{W}_P(b_{rem})$ for $i \in \mathcal{S}_{rem}$ such that $i \leq b_{rem}$ (Line 10). If there is no boundary b_{rem}, then $\mathcal{W}_P(i) = \emptyset$ for all $i \in \mathcal{S}_{rem}$. We do nothing in that case.

In Algorithm 16, the invariant is as follows. For $i \in \mathcal{S}_{rem}$, $\mathcal{W}_P(i) \neq \emptyset$ if $i \leq l \cdot p_k + rem$. For $i \in \mathcal{S}_{rem}$, $\mathcal{W}_P(i) = \emptyset$ if $i \geq r \cdot p_k + rem$. Each condition check of the binary

Algorithm 16: Finalize $i \in \mathcal{T}_{k+1} \backslash \mathcal{T}_k$ s.t. $i \equiv rem \pmod{p_k}$.

```
1  Function Finalize(tail, old_tail, p, rem)
2      l ← ⌈(tail − rem)/p⌉ − 1,  r ← ⌊(old_tail − 1 − rem)/p⌋ + 1;
3      while r − l > 1 do
4          i ← ⌊(l + r)/2⌋,  j ← i · p + rem;
5          if CheckParallel(P̃[1 : m − j], P̃_{j+1}[1 : m − j]) = 0 then r ← i ;
6          else l ← i;
7      b_rem ← l · p + rem;
8      w ← CheckParallel(P̃[1 : m − b_rem], P̃_{b_rem+1}[1 : m − b_rem]);
9      for each i ∈ {tail, ..., b_rem} do in parallel
10         if W[i] = 0 and i ≡ b_rem (mod p) then W[i] ⇐ w + b_rem − i;
```

search (Line 5) takes $O(\xi^{\mathrm{t}}_m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m)$ work. Thus, the overall complexity of Algorithm 16 is $O(\xi^{\mathrm{t}}_m \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m \log m)$ work.

Lemma 4.12. *In round k, Algorithm 15 finalizes $Tail_{k+1}$ in $O(\xi^{\mathrm{t}}_m \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m \log m)$ work on P-CRCW PRAM.*

Proof: First, if $|Tail_{k+1}| = |Tail_k| + 2^k$, Algorithm 15 finalizes all positions $i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k$ in $O(1)$ time and $O(m)$ work. Next, let us consider the case when $|Tail_{k+1}| = LCP_P(p_k)$. Algorithm 15 finalizes all positions $i \in \mathcal{S}_{rem}$ such that $W[q_{rem}] \neq 0$ in $O(1)$ time and $O(m)$ work. Considering $i \in \mathcal{S}_{rem}$ such that $W[q_{rem}] = 0$, since $Head_k$ is 2^k-sparse and $2^k \leq p_k < 2^{k+1}$, there are at most three zero positions in the suffix of length p_k of $Head_k$. Therefore, there are at most three rem where $W[q_{rem}] = 0$. Algorithm 15 updates all positions of $\mathcal{S}_{rem}$ in parallel in $O(\xi^{\mathrm{t}}_m \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m \log m)$ work. Thus, overall Algorithm 15 runs in $O(\xi^{\mathrm{t}}_m \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m \log m)$ work. ■

Theorem 4.13. *Given $\widetilde{P}$, the pattern preprocessing Algorithm 14 computes a witness table in $O(\xi^{\mathrm{t}}_m \cdot \log^2 m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m \log^2 m)$ work on the P-CRCW PRAM.*

Proof: When the algorithm halts, by $2^k \leq tail$, the head size is at most 2^k. Therefore, the head is zero-free except for $W[0] = 0$ by the 2^k-sparsity. By the invariant, $W[i] \in \mathcal{W}_P(i)$ for all the positions of the head. On the other hand, every position of the tail is finalized and has a correct value in the witness table.

In Algorithm 14, the while loop runs $O(\log m)$ times, and each loop takes $O(\xi^{\mathrm{t}}_m \log m)$ time and $O(\xi^{\mathrm{t}}_m \cdot m \log m)$ work, by Lemmas 4.10 and 4.12. Thus, the overall complexity of Algorithm 14 is $O(\xi^{\mathrm{t}}_m \cdot \log^2 m)$ time and $O(\xi^{\mathrm{t}}_m \cdot m \log^2 m)$ work. ■

4.2.2 Pattern searching

Pattern searching algorithm prunes candidates in two stages: dueling and sweeping stages. During the dueling stage, candidate positions duel with each other, until the surviving candidate positions are pairwise consistent and no pattern occurrence is eliminated during

the dueling stage. Recall that x is consistent with $x+a$ if either $0 \leq a < m$ and $W[a] = 0$ or $a \geq m$. A set of positions is *consistent* if all elements in the set are pairwise consistent.

During the sweeping stage, the surviving candidates from the dueling stage are further pruned so that only pattern occurrences survive. To keep track of the surviving candidates, we define a Boolean array $C[1{:}m{+}1]$ and initialize every entry of C to *True*. If a candidate T_i gets eliminated, we set $C[i] = False$. The pattern searching algorithm updates C in such way that $C[i] = True$ iff i is a pattern occurrence. Entries of C are updated at most once during the dueling and sweeping stages.

Let us review Vishkin's pattern searching algorithm [44]. Let us suppose w.l.o.g. that m is a power of 2. If the pattern is aperiodic, then Vishkin's algorithm performs $\log m$ rounds of parallel duels w.r.t. the text. Recall that P is aperiodic, if the smallest period $p > 0$ of P satisfies $2p > m$. Vishkin's preprocessing algorithm is based on the following facts. After the parallel duels, every block of C of length $2^{\log m}$ contains at most one candidate left to check. If the pattern is periodic, let $p > 0$ the smallest period of P. First, Vishkin's algorithm find all occurrences of $P' = P[1 : p]$ in T, using the procedure for aperiodic pattern. Then, for each position i such that i is an occurrence of P', Vishkin's algorithm counts how many times P' consecutively appears in T staring from position i. If P' appears at least $\lceil |P|/p \rceil$ times, then i is an occurrence. This algorithm works for the exact matching, because, for the exact matching all block-based periods are also border-based periods. However, for SCER, this property does not hold.

Dueling stage

The dueling stage is described in Algorithm 17. A set of positions is *consistent* if all elements in the set are pairwise consistent. During the round k, the algorithm partitions C into blocks of size 2^k. Let $\mathcal{C}_{k,j} \subseteq \{(j-1)2^k + 1, \ldots, j \cdot 2^k\}$ be the set of candidate positions in the j-th 2^k-block which have survived after the round k. The invariant of Algorithm 17 is as follows.

Algorithm 17: Parallel algorithm for the dueling stage

```
1  Function DuelingStageParallel()
2      for each j ∈ {1, ..., m} do in parallel
3          C_{0,j}[1] ⇐ j;
4      k ← 1;
5      while k ≤ ⌈log m⌉ do
6          for each j ∈ {1, ..., ⌈m/2^k⌉} do in parallel
7              A ← C_{k−1,2j−1}, B ← C_{k−1,2j};
8              ⟨a, b⟩ ← Merge(A, B);
9              Let C_{k,j} be array of length (a + |B| − b + 1);
10             for each i ∈ {1, ..., a} do in parallel
11                 C_{k,j}[i] ⇐ A[i];
12             for each i ∈ {b, ..., |B|} do in parallel
13                 C_{k,j}[a + i − b + 1] ⇐ B[i];
14         k ← k + 1;
15     Initialize all elements of C to False;
16     for each i ∈ {1, ..., |C_{⌈log m⌉,1}|} do in parallel
17         C[C_{⌈log m⌉,1}[i]] ⇐ True;
```

- At any point of execution of Algorithm 17, all pattern occurrences survive.
- For round k, each $\mathcal{C}_{k,j}$ is consistent.

Set $\mathcal{C}_{k,j}$ is obtained by "merging" $\mathcal{C}_{k-1,2j-1}$ and $\mathcal{C}_{k-1,2j}$. That is, $\mathcal{C}_{k,j}$ shall be a consistent subset of $\mathcal{C}_{k-1,2j-1} \cup \mathcal{C}_{k-1,2j}$ which contains all the occurrence positions in $\mathcal{C}_{k-1,2j-1} \cup \mathcal{C}_{k-1,2j}$. After the dueling stage, $\mathcal{C}_{\lceil \log m \rceil,1}$ is a consistent set including all the occurrence positions. We then let $C[i] = \mathit{True}$ iff $i \in \mathcal{C}_{\lceil \log m \rceil,1}$. In our algorithm, each set $\mathcal{C}_{k,j}$ is represented as an integer array, where elements are sorted in increasing order.

Let us consider merging two consistent sets $\mathcal{A}$ and $\mathcal{B}$ where $\mathcal{A}$ precedes $\mathcal{B}$. For two sets of positions $\mathcal{A}$ and $\mathcal{B}$, we say that $\mathcal{A}$ *precedes* $\mathcal{B}$, iff $\max \mathcal{A} < \min \mathcal{B}$. Sets $\mathcal{A}$ and $\mathcal{B}$ should be merged in such way that the resulting set is consistent and all occurrences in $\mathcal{A}$, $\mathcal{B}$ remain in the resulting set. Let us represent elements of $\mathcal{A}$ and $\mathcal{B}$ on a 2D grid G, as shown in Figure 4.4. In Figure 4.4, elements of $\mathcal{A}$ and $\mathcal{B}$ are presented along the directions of rows and columns, respectively. An intersection point of row i and column

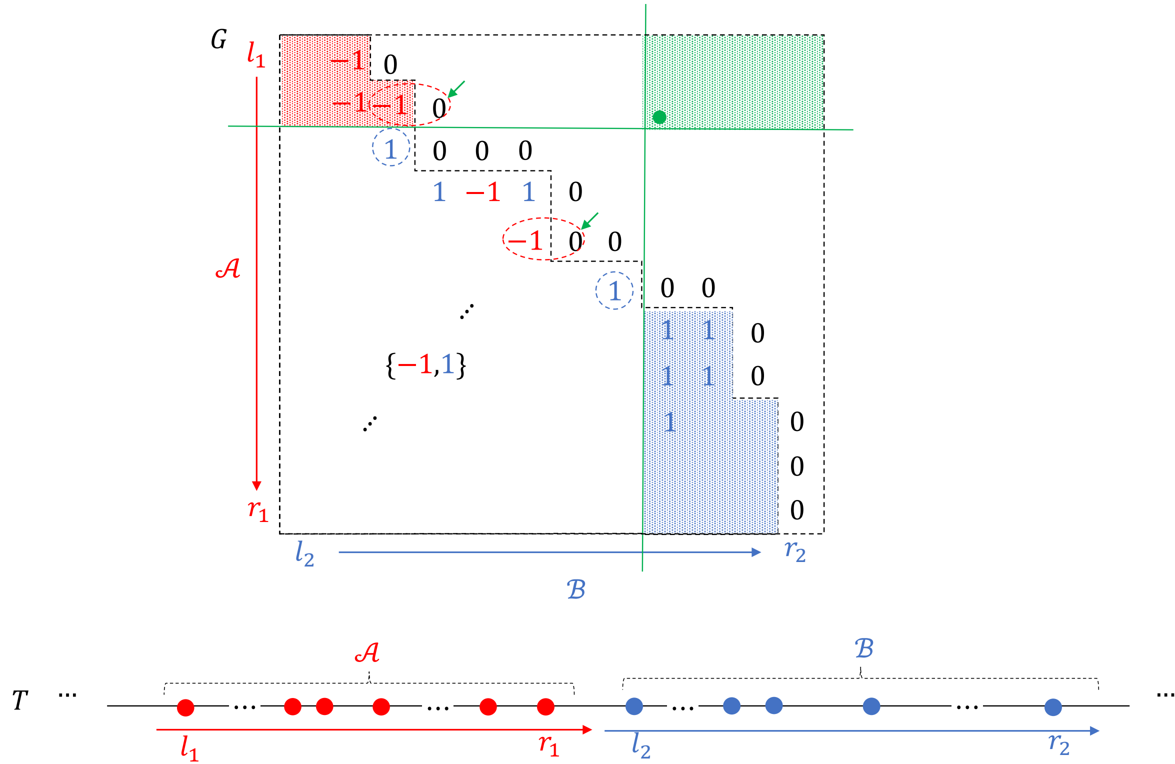


Figure 4.4: Grid G given two consistent sets $\mathcal{A}$ and $\mathcal{B}$.

j is denoted as (i, j), where $i \in \mathcal{A}$ and $j \in \mathcal{B}$, and an element at point (i, j) is denoted as $G[i][j]$. Grid G consists of only -1, 0 and 1. The value of $G[i][j]$ is determined by the value of $W[j - i]$. If $W[j - i] = 0$, then $G[i][j] = 0$. If $W[j - i] \neq 0$ and i wins the duel between i and j, then $G[i][j] = -1$. Else if, j wins the duel $G[i][j] = 1$. We will see that, regardless of the given pattern and witness table, grid G can be divided into two regions: the upper-right region that consists of only 0 and the rest that consist of a mixture of -1 and 1. Throughout this section, $a, a', i \in \mathcal{A}$ and $b, b', j \in \mathcal{B}$, unless stated otherwise. By the consistency property in Lemma 3.1, we can confirm that the following lemma holds.

Lemma 4.14. *Suppose that we are given two position sets $\mathcal{A}$ and $\mathcal{B}$, each of which is consistent and $\mathcal{A}$ precedes $\mathcal{B}$. If $a \in \mathcal{A}$ and $b \in \mathcal{B}$ are consistent, then $\{i \in \mathcal{A} \mid i \leq a\} \cup \{j \in \mathcal{B} \mid j \geq b\}$ is also consistent.*

Proof: Since candidate positions i and a, a and b, b and j are consistent, respectively, by Lemma 3.1, i is consistent with j. ■

From Lemma 4.14, we can see that the region, that consists of only 0, can be separated from the non-zero region with a line that looks like a step function (Figure 4.4). That is, for each row i, there exists some b, such that $G[i][j] = 0$ iff $j \geq b$. Moreover, for two rows i and i', if $i < i'$, then $b \leq b'$ where b' is equivalent of b for row i'.

Let, $i_{\max} = \max\{i \in \mathcal{A} \mid T_i \approx P\}$ and $j_{\min} = \min\{j \in \mathcal{B} \mid T_j \approx P\}$. Visually, given the grid G, whose elements are initially hidden from us, merging $\mathcal{A}$ and $\mathcal{B}$ is equivalent to finding a rectangle in the upper-right corner that consists of 0 and covers all points (i, j) such that $i \leq i_{\max}$ and $j \geq j_{\min}$. In Figure 4.4, the bottom left corner of the green shaded region is located at the point $(i_{\max}, j_{\min})$. By Lemma 4.14, the green region consists of only zeros. Lemma 4.15 shows that the non-zero region to the left of the green region consists of only -1 and the non-zero region under the green region consists of only 1.

Lemma 4.15. *If a is an occurrence and $i \leq a$, then row i consists only of non-positive elements. Similarly, if b is an occurrence and $j \geq b$, then column j consists only of non-negative elements.*

Proof: We prove the first half of the lemma. The second half can be proven in similar manner. Let us consider $i = a$. Since a is a pattern occurrence, for any j that a is not consistent with, a always wins the duel. Thus, if $G[a][j] \neq 0$, then $G[a][j] = -1$. Now, let us consider $i < a$. For any j, $i < a < j$. Since $\mathcal{A}$ is a consistent set and a is a pattern occurrence, i is not consistent with j and i always wins any duel against j. Thus, if $G[i][j] \neq 0$, then $G[i][j] = -1$. ■

We define $\lim_{\mathcal{A}} \in \mathcal{A}$ to be the greatest integer such that for any $i \leq \lim_{\mathcal{A}}$, row i consists only of non-positive elements. Similarly, we define $\lim_{\mathcal{B}} \in \mathcal{B}$ to be the smallest integer such that for any $j \geq \lim_{\mathcal{B}}$, column j consists only of non-negative elements. We have the following corollary from Lemma 4.15.

Corollary 4.16. *If a is an occurrence, then $a \leq \lim_{\mathcal{A}}$. Similarly, if b is an occurrence, then $b \geq \lim_{\mathcal{B}}$.*

Lemma 4.17. *If, for $a \in \mathcal{A}$, $b \in \mathcal{B}$, row $(a+1)$ contains a positive element and column $(b-1)$ contains a negative element, then $a \geq \lim_{\mathcal{A}}$ and $b \leq \lim_{\mathcal{B}}$.*

Proof: Recall that $\lim_{\mathcal{A}}$ is defined to be the greatest integer such that for $i \leq \lim_{\mathcal{A}}$ row i consists only of non-positive elements. Also, $\lim_{\mathcal{B}}$ is defined to be the smallest integer such that for $j \geq \lim_{\mathcal{B}}$ column j consists only of non-negative elements. Since row $(a+1)$ contains a positive element, $(a+1)$ must be greater than $\lim_{\mathcal{A}}$. Since column $(b-1)$ contains a negative element, $(b-1)$ must be less than $\lim_{\mathcal{B}}$. Thus, $a \geq \lim_{\mathcal{A}}$ and $b \leq \lim_{\mathcal{B}}$. ■

Using Corollary 4.16, we can rephrase the goal of our merging algorithm. The goal of the merging algorithm is to find a and b such that $a \geq \lim_{\mathcal{A}}$, $b \leq \lim_{\mathcal{B}}$ and $G[a][b] = 0$. Then $\mathcal{A}_{\leq a} \cup \mathcal{B}_{\geq b}$ satisfies the desired property for merging $\mathcal{A}$ and $\mathcal{B}$. Our merging algorithm is described in Algorithm 18. Given two consistent sets $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{A}$ precedes $\mathcal{B}$, let $l = \max(|\mathcal{A}|, |\mathcal{B}|)$. Naively merging $\mathcal{A}$ and $\mathcal{B}$ will take $O(l^2)$ time on a single processor or $O(1)$ time on $O(l^2)$ processors. By using the properties above, we merge $\mathcal{A}$ and $\mathcal{B}$ in $O(\log^2 l)$ time on a single processor. We achieve this complexity by modifying the 2D binary search serial algorithm. In Algorithm 18, $\mathcal{A}$ and $\mathcal{B}$ are represented as integer arrays, where the elements are sorted in increasing order. For the sake of convenience, we pad grid G with -1s along the leftmost column, with 1s along the bottom row and with 0s along the upper row and rightmost column. Specifically, for $i \in \{0, \ldots, |\mathcal{A}|\}$, $G[i][0] = -1$, for $j \in \{0, \ldots, |\mathcal{B}|\}$, $G[|\mathcal{A}|+1][j] = 1$ and for $i \in \{1, \ldots, |\mathcal{A}|+1\}$, $G[i][|\mathcal{B}|+1] = 0$ and for $j \in \{1, \ldots, |\mathcal{B}|+1\}$, $G[0][j] = 0$. Padding in this manner does not affect the outcome of the Algorithm 18.

At the beginning of each iteration of the outer loop, Algorithm 18 considers row $m_1 = \lfloor (l_1 + r_1)/2 \rfloor$. Given row m_1, the inner loop of Algorithm 18 finds the minimum position m_2 such that $G[m_1][m_2] = 0$ using serial binary search. If 1 is observed in row m_1, Algorithm 18 updates r_1 to m_1. Otherwise, the value of l_1 is set to m_1. Note that the algorithm does not look into every element of row m_1, thus, it is possible that 1 actually

Algorithm 18: Merge two consistent sets $\mathcal{A}$ and $\mathcal{B}$

```
1  Function Merge(A, B)
2      l1 ← 0, r1 ← |A| + 1;
3      while r1 − l1 > 1 do
4          m1 ← ⌊(l1 + r1)/2⌋, observedOne ← False;
5          l2 ← 0, r2 ← |B| + 1;
6          while r2 − l2 > 1 do
7              m2 ← ⌊(l2 + r2)/2⌋;
8              if W[B[m2] − A[m1]] = 0 then r2 ← m2;
9              else
10                 if Dueling(T̃, A[m1], B[m2]) = A[m1] then l2 ← m2 ;
11                 else
12                     observedOne ← True;
13                     break;
14         if observedOne then r1 ← m1;
15         else l1 ← m1;
16     Find b = min{0 < b ≤ |B| + 1 | G[l1][b] = 0} using serial binary search;
17     return ⟨l1, b⟩
```

existed in row m_1, but the algorithm did not observe it. The inner loop terminates as soon as the algorithm observes 1 in row m_1. At any point of Algorithm 18 execution, the following invariant properties hold.

Lemma 4.18. *At any point of Algorithm 18 execution, $G[m_1][l_2] = -1$ and $G[m_1][r_2] = 0$.*

Proof: Let $b = \min\{0 < b \le |\mathcal{B}| + 1 \mid G[l_1][b] = 0\}$. From Lemma 4.14, for $i \in \{0, \ldots, |\mathcal{B}| + 1\}$, $G[m_1][i] \neq 0$ iff $i \ge b$. The inner loop of Algorithm 18 chooses column $m_2 = \lfloor (l_2 + m_2)/2 \rfloor$. If m_1 is consistent with m_2, which means that $G[m_1][m_2] = 0$, r_2 is updated to m_2. If m_1 is not consistent with m_2 and m_1 survives the duel, which means that $G[m_1][m_2] = -1$, then l_2 is updated to m_2. Otherwise, the inner loop immediately terminates. ■

Lemma 4.19. *At any point of Algorithm 18 execution, (1) $G[l_1][b-1] = -1$ where $b = \min\{0 < b \le |\mathcal{B}| + 1 \mid G[l_1][b] = 0\}$ and (2) there exists 1 in row r_1.*

Proof: (1) When $l = 0$ and $b = 1$, since the row 0 is the padding row with $G[0][0] = -1$

and the rest 0, the statement holds. Suppose that the statement is true for the previous iteration of the outer loop. We prove that the statement also holds for the current iteration of the outer loop. If the inner loop of Algorithm 18 observes 1, then the value of l_1 stays unaltered. Now, suppose the inner loop terminates without observing 1. The inner loop terminates when $r_2 = l_2 + 1$. By the invariant of the inner loop, $G[m_1][l_2] = -1$ and $G[m_1][r_2] = 0$. Since the inner loop did not observe 1, the value of l_1 is updated to m_2. Thus, the statement of the lemma holds, after the current iteration of the outer loop.

(2) At the beginning of Algorithm 18, $r_1 = |\mathcal{A}| + 1$. Since row $(|\mathcal{A}| + 1)$ consists of 1s and a 0 at position $(|\mathcal{A}| + 1, |\mathcal{B}| + 1)$, there exists 1 in row $(|\mathcal{A}| + 1)$. As soon as the inner loop of Algorithm 18 observes 1 in row m_1, the inner loop terminates and the value of r_1 is set to m_1. Otherwise, $l_1 \leftarrow m_1$. The outer loop terminates when $r_1 \leftarrow l_1 + 1$. Thus, the invariant conditions hold for l_1 and r_1 at any point of the execution. ■

Let us consider what happens when Algorithm 18 terminates, where $r_1 = l_1 + 1$. Let $a = l_1$ and $b = \min\{0 < b \leq |\mathcal{B}| + 1 \mid G[a][b] = 0\}$. By Lemma 4.19, row $(a + 1)$ contains 1 and $G[a][b] = 0$ and $G[a][b - 1] = -1$. Thus, $a \geq \lim_{\mathcal{A}}$ and $b \leq \lim_{\mathcal{B}}$ and we have found our answer.

Lemma 4.20. *Given a witness table, $\widetilde{P}$, and $\widetilde{T}$, the dueling stage runs in $O(\xi^{\mathrm{t}}_m \log^3 m)$ time and $O(\xi^{\mathrm{w}}_m m \log^2 m)$ work on P-CRCW-PRAM.*

Proof: First, let us consider Algorithm 18. The inner while loop of Algorithm 18 runs in $O(\log m)$ time on a single processor. The outer while loop of Algorithm 18 runs $O(\log m)$ times. Since a duel takes $O(\xi^{\mathrm{w}}_m)$ work, the overall complexity of Algorithm 18 is $O(\xi^{\mathrm{w}}_m \log^2 m)$ work on a single processor. Since the outer loop runs $O(\log m)$ times and each loop takes $O(\xi_m \log^2 m)$ time, the overall time complexity is $O(\xi_m \log^3 m)$. Now, let us look at the work complexity. Since Algorithm 17 takes $O(\xi^{\mathrm{w}}_m \log^2 m)$ work on a single processor, round k takes $O(\xi^{\mathrm{w}}_m m) + \frac{m}{2^k} \cdot O(\xi^{\mathrm{w}}_m \log^2 m)$ work. Since $k \in \{0, \ldots, \lceil \log m \rceil\}$, the overall work complexity is $O(\xi^{\mathrm{t}}_m \cdot m \log^2 m)$. ■

Algorithm 19: Parallel algorithm for the sweeping stage

```
1  Function SweepingStageParallel()
2      create R[1 : m] and initialize elements of R to 0;
3      k ← ⌈log m⌉;
4      while k ≥ 0 do
5          create Cand[0 : ⌊m/2^k⌋] and initialize its elements to −1;
6          for each i ∈ {1, . . . , m} do in parallel
7              if C[i] = True and (i mod 2^k) > 2^(k−1) then Cand[⌊i/2^k⌋] ⇐ i ;
8          for each b ∈ {0, . . . , ⌊m/2^k⌋} do in parallel
9              x ← Cand[b];
10             if x ≠ −1 then
11                 w ← CheckParallel(P̃[R[x] + 1 : m], T̃_x[R[x] + 1 : m]);
12                 if w = 0 then R[x] ⇐ m;
13                 else R[x] ⇐ R[x] + w − 1;
14         for each i ∈ {1, . . . , m} do in parallel
15             x ← Cand[⌊i/2^k⌋];
16             if i ≤ x and R[x] ≤ m − (x − i) − 1 then C[i] ⇐ False ;
17             if i ≥ x and C[i] = True then R[i] ⇐ R[x] − (i − x) ;
18         k ← k − 1;
```

Sweeping stage

The sweeping stage is described in Algorithm 19. The sweeping stage updates C until $C[i] = True$ iff i is a pattern occurrence. All entries in C are updated at most once. Recall that all candidates that survived from the dueling stage are pairwise consistent. At any point of the execution, if $C[i] = False$, then i is not a pattern occurrence. In addition to C, we will create a new integer array $R[1 : m]$. Throughout the sweeping stage, we have the following invariant properties:

- if $C[x] = False$, then $T_x \not\approx P$,
- if $C[x] = True$, then $LCP(T_x, P) \geq R[x]$.

The purpose of bookkeeping this information in R is to ensure that the sweeping stage algorithm uses $O(n)$ processors in each round. Throughout this section, we assume that a processor is attached to each position of C and T.

For each stage k, C is divided into 2^k-blocks. Unlike the preprocessing algorithm, k starts from $\lceil \log m \rceil$ and decreases with each round until $k = 0$. Let us look at each round in more detail. For the b-th 2^k-block of C, let $x_{b,k}$ be the smallest index in the second half of the 2^k-block such that $C[x_{b,k}] = \mathit{True}$. The algorithm finds a position $x_{b,k}$ such that $C[x_{b,k}] = \mathit{True}$ and, by the end of round k, surviving candidates i such that $i < x_{b,k}$ belong to the first half of the 2^k-block in consideration and surviving candidates i such that $i \geq x_{b,k}$ belong to the second half of the block in consideration. Algorithm 19 stores the index $x_{b,k}$ in $\mathit{Cand}[b]$. such that $x_{b,k}$ is the smallest index in the second half of the 2^k-block such that $C[x_{b,k}] = \mathit{True}$. In Algorithm 19, we introduce array $\mathit{Cand}[0 : \lfloor m/2^k \rfloor]$ where $\mathit{Cand}[b] = x_{b,k}$. For each $x_{b,k}$, the algorithm computes $\mathit{LCP}(T_{x_{b,k}}, P)$ exactly and store the value in $R[x_{b,k}]$ on Lines 11–13. Suppose that $\mathit{LCP}(T_{x_{b,k}}, P) < m$, i.e., $T_{x_{b,k}} \not\approx P$ and w is a mismatch position. Since all surviving candidate positions are pairwise consistent, if $T_{x_{b,k}} \not\approx P$, then, any candidate $T_{x_{b,k}-a}$ that "covers" w cannot match the pattern. Generally, we have the following.

Lemma 4.21. *For two consistent candidate positions x and $(x - a)$ such that $a > 0$ and $\mathit{LCP}(T_x, P) \leq m - a - 1$, if x is not an occurrence, then $(x - a)$ is also not an occurrence.*

Proof: Let w be the tight mismatch position for $T_x \not\approx P$, i.e., $w = \mathit{LCP}(T_x, P) + 1$. Since x is consistent with $(x - a)$, $(w + a)$ is a mismatch position for $T_{x-a} \not\approx P$. Thus, $(x - a)$ cannot be a pattern occurrence. ■

Using Lemma 4.21, the algorithm updates C in Lines 15–16. For $x_{b,k}$, after calling `CheckParallel`, $R[x_{b,k}]$ contains the length l of the longest prefixes such that $T_{x,k}[1 : l] \approx P[1 : l]$. Then using Lemma 4.21, Algorithm 19 updates C. On Line 17, the algorithm updates the values of $R[i]$ for indices i in the second half of the block if $C[i] = \mathit{True}$. Since the surviving candidates are pairwise consistent, for candidate positions $(x_{b,k} + a)$ such that $a > 0$, $T_{x_{b,k}+a}[1 : r] \approx P[1 : r]$ where $r = R[x_{b,k}] - a$. In this way, the algorithm maintains the invariant properties. When $k = 0$, all the 2^k-blocks contain just one position and $R[x]$ is set to be exactly $\mathit{LCP}(T_x, P)$ by Lines 11–13, unless $C[x] = \mathit{False}$ at that

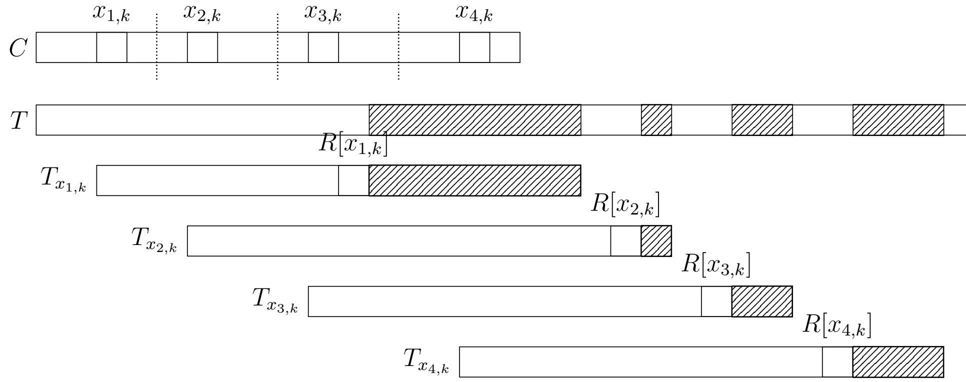


Figure 4.5: Illustration to sweeping stage. The shaded regions of the text are referenced during round k and the referenced regions of T do not overlap.

time. Then, if $R[x] < m$, then $C[x]$ will be *False* on Line 16. That is, when the algorithm halts, $C[x] =$ *True* iff $T_x \approx P$.

It remains to show the efficiency of the algorithm. We can prove that each position of T is referenced at most once during each round. Then we obtain the following lemma. Lemma 4.22 shows that each position of T is referenced at most once during each round. Lemma 4.22 shows that each position of T is referenced at most once during each round.

Lemma 4.22. *After the round k, for two surviving candidate positions i and j with $i < j$ that do not belong to the same 2^{k-1}-block of C, $i + m \leq j + R[j]$.*

Proof: After the round $k = \lceil \log m \rceil + 1$, which is before round $k = \lceil \log m \rceil$, since all candidate positions belong to the same $2^{\lceil \log m \rceil}$-block, the statement holds (base case). Assuming that the statement holds after the round $(k+1)$, we prove that it also holds after the round k. Let R_{k+1} and R_k be the states of the array R after the rounds $(k+1)$ and k, respectively. First, let us consider the case when surviving candidate positions i and j do not belong to the same 2^k-block of C. Obviously, i and j cannot belong to the same 2^{k-1}-block. By the induction hypothesis, $i + m \leq j + R_{k+1}[j]$. Since $R_k[j] \geq R_{k+1}[j]$, $i + m \leq j + R_k[j]$.

Now, let us consider the case when candidate positions i and j belong to the same

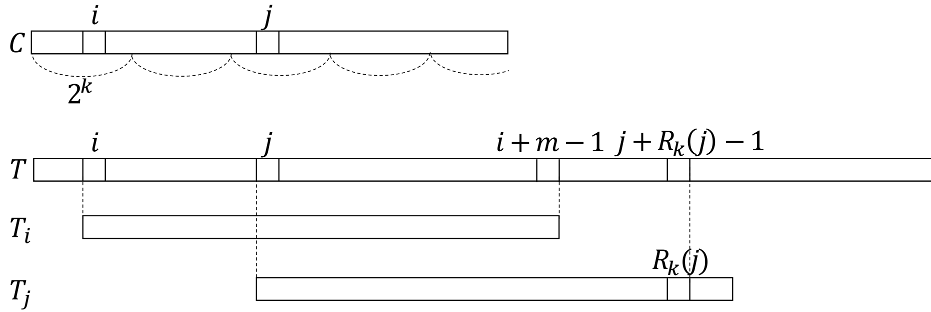


Figure 4.6: Before round k, for two surviving candidates T_i and T_j such that $j - i \geq 2^k$, $i + m - 1 < j + R_k[j]$.

2^k-block of C. During round k, for each 2^k-block of C, Algorithm 19 chooses as surviving candidate position $x_{b,k}$ which is the smallest index in the second half of the 2^k-block. Thus, two surviving candidates positions i and j of the b-th 2^k-block belong to different 2^{k-1}-blocks iff $i < x_{b,k} \leq j$. For T_i to be a surviving candidate after round k, it must be the case that $m + i \leq LCP(T_{x_{b,k}}, P) + x_{b,k}$. For T_j, Algorithm 19 updates $R_k[j]$ to $LCP(T_{x_{b,k}}, P) - (j - x_{b,k})$. Substituting it into the previous inequality, we get $m + i \leq R_k[j] + (j - x_{b,k}) + x_{b,k} = R_k[j] + j$. ■

Lemma 4.23. *Given $\widetilde{P}$ and $\widetilde{T}$, the sweeping stage algorithm finds all pattern occurrences in $O(\xi^{\mathrm{t}}_m \log m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m \log m)$ work on the P-CRCW PRAM.*

Proof: The outer loop of Algorithm 19 runs $O(\log m)$ times. Since the processors that are attached to T are used at most once by Lemma 4.22, each loop runs in $O(\xi^{\mathrm{t}}_m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m)$ processors. Thus, the total time is $O(\xi^{\mathrm{t}}_m \cdot \log m)$ and total work is $O(\xi^{\mathrm{w}}_m \cdot m \log m)$.

By Theorem 4.13 and Lemmas 4.20, and 4.23, we obtain the main theorem. When $n \geq 2m$, T is cut into overlapping pieces of length $(2m - 1)$ and each piece is processed independently.

Theorem 4.24. *Given a witness table, $\widetilde{P}$, and $\widetilde{T}$, the pattern searching solves the pattern searching problem under SCER in $O(\xi^{\mathrm{t}}_m \cdot \log^3 m)$ time and $O(\xi^{\mathrm{w}}_m \cdot n \log^2 m)$ work on the P-CRCW PRAM.*

Algorithm 20: Returns a tight/suffix-tight witness for offset a

```
1 Function TightWitness(a)
2     w ← CheckParallel(P[1 : m − a], P[a + 1 : m], P̃);
3     return w;
4 Function SuffixTightWitness(a)
5     w ← CheckParallel(P^R[1 : m − a], P^R[a + 1 : m], P̃^R);
6     w ← m − w − a + 1;
7     return w;
```

4.3 Reversible SCER

In this section, we will consider SCERs that are *reversible*. That is, given two strings X and Y, $X \approx Y$ iff $X^R \approx Y^R$. Recall that X^R is the reverse of string X. For such SCER, a witness table construction can be performed in $O(\xi_m^{\mathrm{t}} \cdot \log m)$ time and $O(\xi_m^{\mathrm{w}} \cdot m \log m)$ work on the P-CRCW PRAM, which improves the preprocessing algorithm for the general SCER by a factor of $\log m$. This improvement comes from the fact that, for each round, tail finalization for the witness table can be performed in $O(\xi_m^{\mathrm{t}} \cdot 1)$ time and $O(\xi_m^{\mathrm{w}} \cdot m)$ work. The algorithm for pattern searching is same as the algorithm for the general SCER.

We define $LCS_X(a)$ to be the length of the longest common suffix, when X is superimposed on itself with offset a. Formally, given an integer $0 \leq a < |X|$, $LCS_X(a) = l$ is the greatest integer such that $X[m - l + 1 : m] \approx X[m - a - l + 1 : m - a]$ and $X[m-l:m] \not\approx X[m-a-l:m-a]$. Since, for two strings X and Y, $X \approx Y \Leftrightarrow X^R \approx Y^R$, $LCS_P(a) = LCP_{P^R}(a)$ for any offset a.

Recall that for a witness w for offset a, if $w = LCP_P(a) + 1$, then we called w a *tight* witness for offset a. Analogously, we define *suffix-tight* witness for offset a. A witness w is a *suffix-tight* witness, if $w = m - a - LCS_P(a)$.

Given encoding of P^R, $LCP_{P^R}(a)$ can be computed in $O(1)$ time on $O(m)$ processors using `CheckParallel`. Let w the value returned by `CheckParallel`, which is the tight mismatch position for $P^R[1 : m - a] \not\approx P^R[a + 1 : m]$. Since $LCS_P(a) = LCP_{P^R}(a)$, we have the following. If $w = 0$, then $LCP_{P^R}(a) = LCS_P(a) = m - a$. If $w \neq 0$,

Algorithm 21: Finalize $i \in \mathcal{T}_{k+1} \backslash \mathcal{T}_k$ s.t. $i \equiv rem \pmod{p_k}$ (reversible SCER).

```
1 Function Finalize(tail, old_tail, p, rem)
2     r_min ← min{r | tail ≤ r < old_tail and r ≡ R[i] (mod p)};
3     w ← SuffixTightWitness(r_min);
4     LCS ← m − r_min − w − 1;
5     for each i ∈ {tail, ..., old_tail − 1} do in parallel
6         if i ≡ r_min (mod p_k) and m − i > LCS then
7             W[i] ← w − (i − r_min);
```

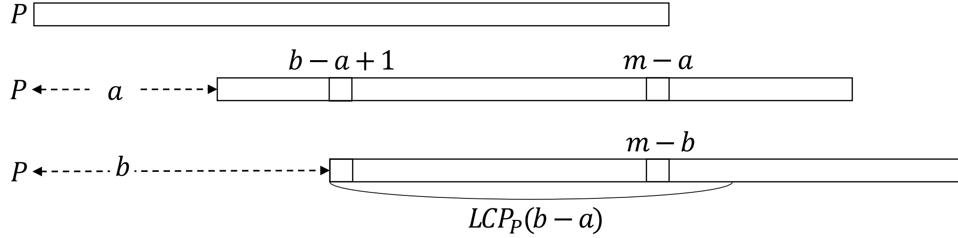


Figure 4.7: For $a, b \in Tail_k$ such that $a < b$ and $a \equiv b \pmod{p_k}$, $P[b-a+1:m-a] \approx P[1:m-b]$

then $LCP_{P^R}(a) = LCS_P(a) = w - 1$. Since w is the tight mismatch position pair for $P^R[a+1:m] \not\approx P^R[1:m-a]$, after reversing the indices for $P[a+1:m]$ and $P[1:m-a]$, $(m-w-a+1)$ is the suffix-tight witness for offset a. The procedure is described in Algorithm 20.

Next, we discuss how the algorithm finalizes $Tail_{k+1}$ for the round k. This procedure is described in Algorithm 21. Recall that we have the following property for positions of $Tail_k$. For round k, given positions $a, b \in Tail_{k+1}$ such that $a < b$ and $a \equiv b \pmod{p_k}$, $P[b-a+1:m-a] \approx P[1:m-b]$ (Figure 4.7). This stays same as the SCER general case algorithm.

For $i \in \{Tail_{k+1} \backslash Tail_k\}$, let $r_{\max} = \max\{r \in Tail_{k+1} \backslash Tail_k \mid i \equiv r \pmod{p_k}\}$. Depending on whether the algorithm found a witness for offset $r_{\max}$ in prior rounds, Algorithm 15 takes different approach to finalize $W[i]$ (Figure 4.9). If $W[r_{\max}] \neq 0$, then from Lemma 4.11 we see that $W[r_{\max}] + (r_{\max} - i) \in \mathcal{W}_P(i)$. Now, let us consider $i \in \{Tail_{k+1} \backslash Tail_k\}$ such that $W[r_{\max}] = 0$. This case is more involved than the previous

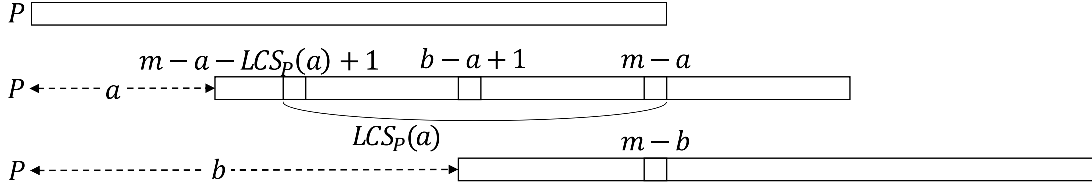


Figure 4.8: Illustration to Lemma 4.25.

case, but the algorithm uses the following lemma to finalize such $W[i]$ efficiently.

Lemma 4.25. *For round k, given offsets $a, b \in Tail_{k+1}$ such that $a < b$, $\mathcal{W}_P(b) = \emptyset$, iff $m - b \leq LCS_P(a)$ (Figure 4.8). If $\mathcal{W}_P(b) \neq \emptyset$, then $w - (b - a) \in \mathcal{W}_P(b)$ where w is a suffix-tight witness for offset a.*

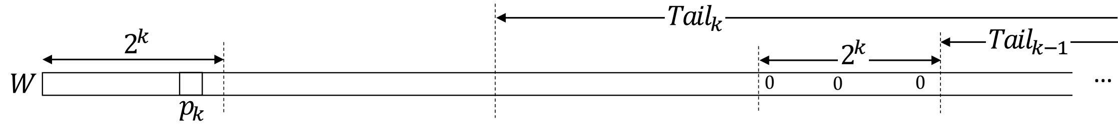
Proof: First, we prove the first half of the lemma. $\mathcal{W}_P(b) = \emptyset$ means that $P[1 : m - b] \approx P[b + 1 : m]$. Furthermore, by Lemma 4.11, $P[1 : m - b] \approx P[b - a + 1 : m - a]$. It means that $LCS_P(a)$ must be greater than or equal to $m - b$. Now, we will prove the converse. By the definition of $LCS_P(a)$, $P[m - a - LCS_P(a) + 1 : m - a] \approx P[m - LCS_P(a) + 1 : m]$. Since, $m - b \leq LCS_P(a)$, $\mathcal{W}_P(b) = \emptyset$.

Now, we prove the second half of the lemma. Recall that w is a suffix-tight witness for offset a, if $w = m - a - LCS_P(a)$. Since $m - b > LCP_P(a)$ and $P[1{:}m - b] \approx P[b - a + 1{:}m - a]$ by Lemma 4.11, $w - (b - a) \in \mathcal{W}_P(b)$. ■

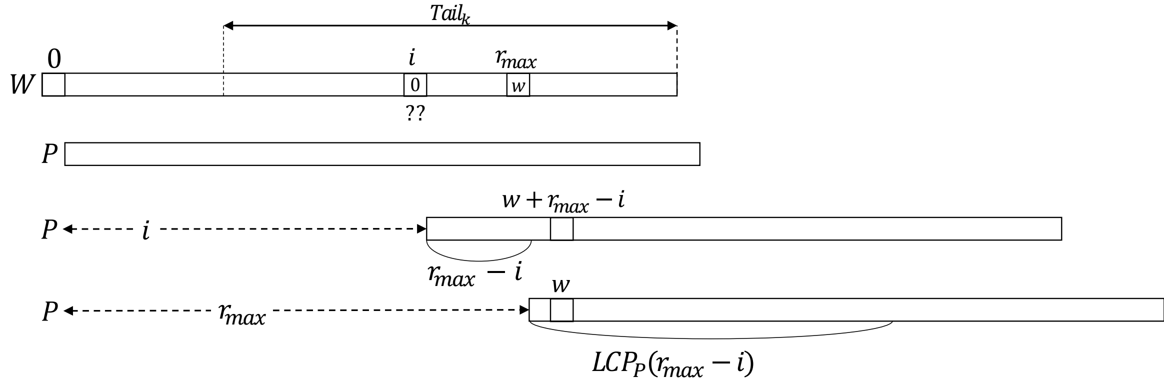
For $i \in \{Tail_{k+1} \backslash Tail_k\}$ such that $W[r_{\max}] = 0$, Algorithm 21 finalizes such positions. For such i, let $r_{\min} = \min\{r \in Tail_k \backslash Tail_{k-1} \mid i \equiv r \pmod{p_k}\}$. Algorithm 21 first computes $LCS_P(r_{\min})$ and a suffix-tight witness w for offset $r_{\min}$. Then, using Lemma 4.25, for $i \in \{Tail_{k+1} \backslash Tail_k\}$ such that $m - i > LCS_P(r_{\min})$, the algorithm updates $W[i]$ to $w - (i - r_{\min})$.

Lemma 4.26. *For the round k, Algorithm 21 finalizes $Tail_k$ in $O(\xi_m^{\mathrm{t}})$ time and $O(\xi_m^{\mathrm{w}} \cdot m)$ work on P-CRCW PRAM.*

Proof: First, Algorithm 15 finalizes all positions $i \in \{Tail_{k+1} \backslash Tail_k\}$ such that $W[r_{\max}] \neq 0$ in $O(1)$ time and $O(m)$ work. Next, let us consider $i \in \{Tail_{k+1} \backslash Tail_k\}$ such that



Case 1. $W[r_{max}] \neq 0$



Case 2. $W[r_{max}] = 0$

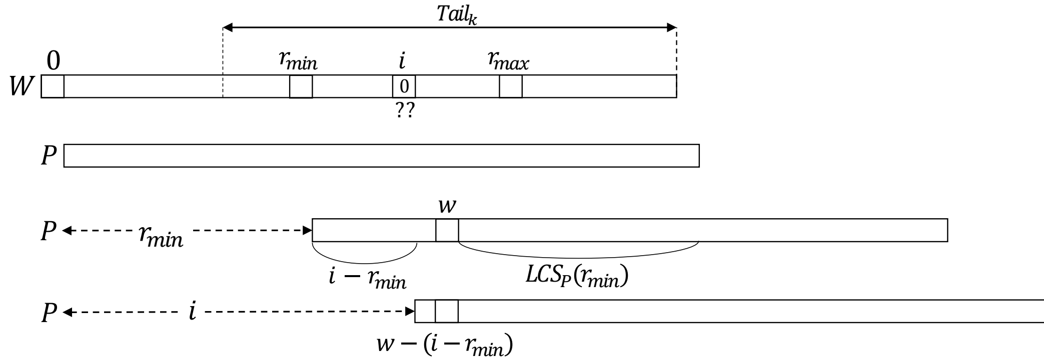


Figure 4.9: Illustration to Algorithm 21 for round k.

$W[r_{\max}] \neq 0$. Since $W[0 : |Head_{k-1}| - 1]$ is 2^{k-1}-sparse and $2^{k-1} \leq p_k < 2^k$, there are at most three zero positions in $W[m - |Tail_k| - p_k : m - |Tail_k| - 1]$. Therefore, there are at most three such $r_{\max}$. For each such $r_{\max}$, the algorithm computes $LCS_P(r_{\min})$ in $O(1)$ time and $O(m)$ work. Then, it finalizes positions $i \in \{Tail_{k+1} \backslash Tail_k\}$ such that $W[r_{\max}] = 0$ in $O(1)$ time and $O(m)$ work. Thus, overall Algorithm 15 runs in $O(1)$ time and $O(m)$ work. ∎

Theorem 4.27. *The pattern preprocessing for reversible SCER can be performed in* $O(\xi_m^{\mathrm{t}} \cdot$

$\log m)$ *time and* $O(\xi_m^{\mathrm{w}} \cdot m \log m)$ *work on the P-CRCW PRAM.*

4.4 Parameterized matching

First, we discuss how to compute $prev(X)$ in parallel for a string X. In order to compute $prev(X)$ we will use *all nearest smaller value* problem. All nearest smaller value problem is defined as follows. Let X be an array of elements from a totally ordered domain. For each $1 \leq i \leq |X|$, all nearest smaller value problem finds the maximal $1 \leq j < i$ and the minimal $i < k \leq |X|$ such that $X[j] < X[i]$ and $X[k] < X[i]$.

Without loss of generality, we assume that Π forms a totally ordered domain. Recall that Π is an alphabet of parameter symbols and Σ is an alphabet of constant symbols. We will construct the following string X' from X. We define a new symbol, say ∞, such that, for any element $\pi \in \Pi$, π is less than ∞. For $1 \leq i \leq |X|$, $X'[i] = X[i]$ if $X[i] \in \Pi$ and $X'[i] = \infty$ if $X[i] \in \Sigma$. For X', we construct $Lmax_{X'}$, which was also used in order-preserving matching.

$$Lmax_{X'}[i] = j \quad \text{if} \quad X'[j] = \max_{k<i}\{X'[k] \mid X'[k] \leq X'[i]\}.$$

We use the rightmost (largest) j if there exist more than one such $j < i$. If there is no such j, then we define $Lmax_{X'}[i] = 0$. Suppose that $X[i] \in \Pi$ for $1 \leq i \leq |X|$. After computing $Lmax_{X'}$, $prev(X)[i] = i - Lmax_{X'}[i]$ if $X[i] = X[Lmax_{X'}[i]]$. If $Lmax_{X'}[i] = 0$ or $X[i] \neq X[Lmax_{X'}[i]]$, then $X[i]$ is the first occurrence of this letter. For example, given $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Pi = \{\mathtt{u}, \mathtt{v}\}$, $X = \mathtt{uvuvauuvb}$ let us compute $prev(X)$. Suppose that $u < v$. First, we obtain $X' = \mathtt{uvuv}\infty\mathtt{uuv}\infty$. Then, $Lmax_{X'} = (0, 1, 2, 1, 0, 3, 6, 4, 5)$. Computing $prev(X)$ from $Lmax_{X'}$, we obtain $prev(X) = 0013\mathtt{a}314\mathtt{b}$.

Lemma 4.28. *Given a string* X *of length* n*,* $Lmax_X$ *can be computed in* $O(\log n)$ *time and* $O(n \log n)$ *work on the P-CRCW PRAM.*

Proof: Following the construction of $Lmax_X$ and $Lmin_X$ by [37], suppose that positions

of X are sorted with respect to their contents. In case of equal contents the smaller positions come first. Let X' be the resulting sequence of positions. For $i \in \{1, \ldots, n\}$, let j be the position of i in X'. Then $Lmax_X[i]$ is the nearest smaller value in X' to the left of $X'[j]$. If there is no such value, $Lmax_X[i] = 0$. Using the merge sort algorithm by Cole [18] and the all-smaller-nearest-values algorithm by Berkman et al. [7], $Lmax_X$ and $Lmin_X$ are computed in $O(\log n)$ time and $O(n \log n)$ work on the P-CRCW PRAM. ■

Now, we discuss our parallel algorithm for pattern matching. Given a witness table, dueling and sweeping stages can be used for parameterized matching by straightforward substitution. Since, given $prev(P)[i]$, $prev(P)_k[i]$ can be computed in $O(1)$ time on a single processor, we have the following. By Theorem 4.24, given a witness table, the pattern searching runs in $O(\log^3 m)$ time and $O(n \log^2 m)$ work on the P-CRCW PRAM.

For the witness table construction, by Theorem 4.13, straightforward substitution will result in an algorithm that runs in $O(\log^2 m)$ time and $O(m \log^2 m)$ work on the P-CRCW PRAM. The witness table can be constructed in $O(\log m)$ time and $O(m \log m)$ work on the P-CRCW PRAM.

4.5 Cartesian-tree matching

Recall that for a string X, its parent-distance encoding [41] for cartesian-tree matching PD_X is defined as follows.

$$PD_X[i] = \begin{cases} i - \max_{1 \leq j < i}\{j \mid X[j] \leq X[i]\} & \text{if such } j \text{ exists,} \\ 0 & \text{otherwise.} \end{cases}$$

Theorem 4.29. *Given a string X of length n, PD_X can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM. Moreover, given PD_X, $PD_{X[x:n]}[i]$ can be computed in $O(1)$ time and $O(1)$ work.*

Proof: For $1 \leq i \leq n$, $PD_X[i]$ is the nearest smaller value to the left of $X[i]$. Since the

all-smaller-nearest-value problem can be solved in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM by Berkman et al. [7], PD_X can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM.

Given PD_X, $PD_{X[x:n]}[i]$ can be computed in the following manner in $O(1)$ time and $O(1)$ work.

$$PD_{X[x:n]}[i] = \begin{cases} 0 & \text{if } PD_X[x+i-1] \geq i, \\ PD_X[x+i-1] & \text{otherwise.} \end{cases}$$

■

Parent-distance encoding is an $\approx$-encoding. Cartesian-tree is not a reversible SCER. Thus, by simple extension of the SCER algorithm for the general case we obtain the following.

Theorem 4.30. *The pattern preprocessing A witness table for cartesian-tree matching can be computed in $O(\log^2 m)$ time and $O(m \log^2 m)$ work on the P-CRCW PRAM.*

Theorem 4.31. *Given a witness table, the pattern searching for cartesian-tree matching can be solved in $O(\log^3 m)$ time and $O(n \log^2 m)$ work on the P-CRCW PRAM.*

4.6 Palindrome matching

In this section, we show a parallel version of the duel-and-sweep algorithm for palindrome matching. Recall that given a string X, let $Pals_X = \{(c, r) \mid X[c-r+0.5 : c+r-0.5]$ is a maximal palindrome at center $c = 1, 1.5, 2, \ldots, |X|\}$ Recall that given two strings X and Y of same length, $X \approx Y$ if $Pals_X = Pals_Y$. For string X, $Pals_X$ does not satisfy Conditions (1) and (2), but it satisfies Conditions (3) and (4) of Definition 2.3.

Lemma 4.32. *([4]) Given a string X of length n, $Pals_X$ can be computed in $O(\log n)$ time and $O(n \log n)$ work on the P-CRCW PRAM.*

Algorithm 22: Returns $LCP(X, Y)$ in parallel

```
1  Function LCPPalindromePar(X, Y, Pals_X, Pals_Y)
2      n ← |X|;
3      create array of integers R[1, 1.5, 2, ..., n];
4      Initialize all elements of R to n;
5      for each c ∈ {1, 1.5, 2, ..., n} do in parallel
6          r_X ← Pals_X[c], r_Y ← Pals_Y[c];
7          if r_X ≠ r_Y then
8              R[c] ⇐ c + min(r_X, r_Y) − 0.5;
9      Let j be the smallest element of R;
10     return j;
```

Algorithm 23: Returns a tight mismatch center if $X \not\approx Y$

```
1  Function CheckParallel(X, Y, Pals_X, Pals_Y)
2      w ← 0;
3      lcp ← LCPPalindromePar(X, Y, Pals_X, Pals_Y);
4      if lcp < n then
5          for each c ∈ {1, 1.5, 2, ..., |X|} do in parallel
6              r_X ← Pals_{X[1:lcp+1]}[c], r_Y ← Pals_{Y[1:lcp+1]}[c];
7              if |r_X − r_Y| = 1 then
8                  w ⇐ c;
9      return w;
```

First, we define a *witness* for parallel parameterized pattern matching. For a center c and an offset a, let $(c, r_{pref}) \in Pals_{P[1:m]}$ and $(c, r_{suff}) \in Pals_{P[a+1:m]}$. An integer c is a witness for offset a, if $r_{pref} \neq r_{suff}$. For two mismatching strings X and Y of same length, a center c is called a *tight mismatch center* if $|r_X - r_Y| = 1$ where $(c, r_X) \in Pals_{X[1:LCP(X,Y)+1]}$ and $(c, r_Y) \in Pals_{Y[1:LCP(X,Y)+1]}$. A witness c is a *tight* witness if c is a tight mismatch center for $P[a+1 : m] \not\approx P[1 : m]$.

Next, we show how to compute $LCP(X, Y)$ of two strings X and Y in parallel. The procedure is shown in Algorithm 23.

Lemma 4.33. *For strings X and Y of equal length n, Algorithm 22 computes $LCP(X, Y)$ in $O(1)$ time and $O(n)$ work on the P-CRCW PRAM, assuming that $Pals_X$ and $Pals_Y$ are already computed.*

Proof: First, we prove the correctness of the algorithm. If $X \approx Y$, then $R[c] = n$ for all centers c. Thus, the algorithm retuns $n = LCP(X, Y)$ where $n = |X|$. If $x \not\approx Y$, then there must exist a center c such that $r_X \neq r_Y$ where $(c, r_X) \in Pals_X$ and $(c, r_Y) \in Pals_Y$. $R[c]$ contains the maximal position i for which $Pals_{X[1:i]}(c) = Pals_{Y[1:i]}(c)$. If j be the minimum value in R at the end of the algorithm, then $X[1 : j]$ is the maximal prefix such that $X[1 : j] \approx Y[1 : j]$, i.e., $LCP(X, Y) = j$.

Next, we prove the computational complexity of the algorithm. Since the minimum element of R can be found in $O(1)$ time and $O(n)$ work on P-CRCW PRAM, the overall complexity of the algorithm is $O(1)$ time and $O(n)$ work on P-CRCW PRAM. ■

For strings X and Y of same length n, Algorithm 23 finds a tight mismatch center in $O(1)$ time and $O(n)$ work on P-CRCW PRAM. If $X \approx Y$, then Algorithm 23 returns 0.

We discuss dueling w.r.t. the text. If $w \in \mathcal{W}_P(a)$, then it holds that

- if $Pals_{T_{x+a}}(w) = Pals_P(w)$, then $T_x \not\approx P$,
- if $Pals_{T_{x+a}}(w) \neq Pals_P(w)$, then $T_{x+a} \not\approx P$.

Next, we discuss dueling w.r.t. the pattern. For convenience, for a witness w for offset a, we denote *witness radius* $w_r = \min(Pals_{P[a+1:m]}(w), Pals_{P[1:m-a]}(w)) + 0.5$. For a witness w for offset a, $w + w_r - 0.5 > LCP_P(a)$. If w is a tight mismatch center for offset a, then $w + w_r - 0.5 = LCP_P(a) + 1$. Suppose $w \in \mathcal{W}_P(a)$, $j \leq m - (w + w_r - 0.5)$ and $j - i = a$. Then,

1. if the offset j survives the duel, i.e., $Pals_{P[j+1:m]}(a) = Pals_{P[1:m-j]}(a)$, then $w + a \in \mathcal{W}_P(i)$;
2. if the offset i survives the duel, i.e., $Pals_{P[j+1:m]}(a) = Pals_{P[a+1:m-i]}(a)$, then $w \in \mathcal{W}_P(j)$.

Given a witness table, dueling w.r.t. T and P is performed in $O(1)$ time and $O(1)$ work on P-CRCW PRAM. For round k, suppose the preprocessing invariant holds true and

$\mathcal{W}_P(p_k) \neq \emptyset$. Let us consider any two positions i, j of $Head_{k+1}$ such that $0 < j - i < 2^{k+1}$. In the discussions about the SCER algorithm, we have shown that $j < m - LCP_P(j-i)$. Since for any $w \in \mathcal{W}_P(j-i)$, $w + w_r - 0.5 > LCP_P(j-i)$, $j \leq m - (w + w_r - 0.5) < m - LCP_P(j-i)$. Thus, the 2^{k+1}-sparsity of $Head_{k+1}$ is satisfied in $O(1)$ time and $O(m/2^k)$ work on P-CRCW PRAM. Suppose $m - LCP_P(p) \leq b < m$. Since $Pals_P$ satisfies Condition (3) of Definition 2.3, we have the following. If $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$ for any offset a such that $0 \leq a \leq b$ and $a \equiv b \pmod{p}$.

Lemma 4.34. *Given $Pals_P$, a witness table for palindrome matching can be computed in $O(\log m)$ time and $O(m \log m)$ work on P-CRCW PRAM.*

As for the pattern searching, using the discussions above, we can extend the SCER algorithm to obtain the following.

Theorem 4.35. *There exists a parallel algorithm that solves the palindrome pattern matching for in $O(\log^3 m)$ time and $O(n \log^2 m)$ work on the P-CRCW PRAM.*

4.7 Order-preserving matching

In this section, we show a parallel version of the duel-and-sweep algorithm for OPPM. First, we discuss how to compute $Lmax_X$ and $Lmin_X$ in parallel.

Lemma 4.36. *Given a string X of length n, $Lmax_X$ and $Lmin_X$ can be computed in $O(\log n)$ time and $O(n \log n)$ work on the P-CRCW PRAM.*

Proof: Following the construction of $Lmax_X$ and $Lmin_X$ by [37], suppose that positions of X are sorted with respect to their contents. In case of equal contents the smaller positions come first. Let X' be the resulting sequence of positions. For $i \in \{1, \ldots, n\}$, let j be the position of i in X'. in other words, $X'[j] = X[i]$. Then $Lmax_X[i]$ is the nearest smaller value in X' to the left of $X'[j]$. If there is no such value, $Lmax_X[i] = 0$.

Algorithm 24: Check order-isomorphism of X and Y in parallel

```
1 Function CheckParallel(X, Y, Lmax_X, Lmin_X)
2   ⟨w1, w2⟩ ← ⟨0, 0⟩;
3   for each i ∈ {1, ..., |X|} do in parallel
4     i_min ← Lmin_X[i], i_max ← Lmax_X[i];
5     if i_min ≠ 0 and Y[i_min] < Y[i] then
6       ⟨w1, w2⟩ ⇐ ⟨i_min, i⟩;
7     else if i_max ≠ 0 and Y[i_max] > Y[i] then
8       ⟨w1, w2⟩ ⇐ ⟨i_max, i⟩;
9   return ⟨w1, w2⟩;
```

$Lmin_X$ is computed similarly. Using the merge sort algorithm by Cole [18] and the all-smaller-nearest-values algorithm by Berkman et al. [7], $Lmax_X$ and $Lmin_X$ are computed in $O(\log n)$ time and $O(n \log n)$ work on the P-CRCW PRAM. ■

Next, we show how to compute order-isomorphism of two strings in parallel. The procedure is shown in Algorithm 24. Recall that in OPPM, the order-isomorphism of two strings cannot be determined by comparing a symbol in one position. We need two positions to say that the two strings are not order-isomorphic. We say that a mismatch position pair $\langle i, j \rangle$ is *tight* if $X[1 : j-1] \approx Y[1 : j-1]$ and $X[1 : j] \not\approx Y[1 : j]$.

Lemma 4.37. *For strings X and Y of equal length n, Algorithm 24 computes a tight mismatch position pair in $O(1)$ time and $O(n)$ work on the P-CRCW PRAM, assuming that $Lmax_X$ and $Lmin_X$ are already computed.*

Proof: In Algorithm 24, for each element of X, we “attach” a processor to compute $F(X, Y, i)$ defined in Equation 3.6, where i is the position of the element. It can be done in $O(1)$ time because $Lmax_X$ and $Lmin_X$ are given. If $F(X, Y, i) \neq 0$ for some i, the corresponding processor tries to update the shared variable $\langle w_1, w_2 \rangle$ to $\langle i_{\min}, i \rangle$ or $\langle i_{\max}, i \rangle$. In P-CRCW PRAM, the processor with the lowest index will succeed in writing $\langle w_1, w_2 \rangle$ properly. Thus, at the end of the algorithm $\langle w_1, w_2 \rangle$ contains a tight mismatch position pair. ■

Algorithm 25: Returns a tight/suffix-tight witness for offset a

```
1 Function TightWitness(a)
2     ⟨w1, w2⟩ ← CheckParallel(P[1 : m − a], P[a + 1 : m], Lmax_P, Lmin_P);
3     return ⟨w1, w2⟩;
4 Function SuffixTightWitness(a)
5     ⟨w1, w2⟩ ← CheckParallel(P^R[1 : m − a], P^R[a + 1 : m], Lmax_{P^R}, Lmin_{P^R});
6     ⟨w1, w2⟩ ← ⟨m − w1 − a + 1, m − w2 − a + 1⟩;
7     return ⟨w1, w2⟩;
```

Pattern preprocessing

For the rest of this section, we assume that, for a witness pair $\langle i, j\rangle$, $i < j$. For a witness pair $\langle i, j\rangle$ for offset a, if $j = LCP_P(a) + 1$, then we call $\langle i, j\rangle$ a *tight* witness for offset a. Analogously, we define *suffix-tight* witness for offset a. A witness pair $\langle i, j\rangle$ is a *suffix-tight* witness, if $i = m - a - LCS_P(a)$. Given a tight witness $\langle i, j\rangle$ for offset a, $LCP_P(a) = j - 1$. Given a suffix-tight witness $\langle i, j\rangle$ for offset a, $LCS_P(a) = m - a - i - 1$.

We also define binary operations $\oplus$ and $\ominus$ for a position pair $\langle i, j\rangle$ and an integer k. Specifically, $\langle i, j\rangle \oplus k = \langle i + k, j + k\rangle$ and $\langle i, j\rangle \ominus k = \langle i - k, j - k\rangle$. Also, for an integer pair $\langle i, j\rangle$, we denote $\max\langle i, j\rangle = \max\{i, j\}$ and $\min\langle i, j\rangle = \min\{i, j\}$.

Next, we discuss how the algorithm finalizes $Tail_k$ for the round k. For the sake completeness, we state the algorithm for finalizing $Tail_k$ for OPPM for round k in Algorithm 15. For $i \in \{Tail_{k+1} \backslash Tail_k\}$, let $r_{\max} = \max\{r \in Tail_{k+1} \backslash Tail_k \mid i \equiv r \pmod{p_k}\}$. If $W[r_{\max}] \neq \langle 0, 0\rangle$, then, from Lemma 4.11 we see that $W[r_{\max}] \oplus (r_{\max} - i) \in \mathcal{W}_P(i)$. Using similar reasoning as in Lemma 4.25 we have the following.

Lemma 4.38. *If $\mathcal{W}_P(b) \neq \emptyset$, then $\langle w_1, w_2\rangle \ominus (b - a) \in \mathcal{W}_P(b)$ where $\langle w_1, w_2\rangle$ is a suffix-tight witness for offset a.*

Proof: Recall that $\langle w_1, w_2\rangle$ is a suffix-tight witness for offset a, if $w_1 = m - a - LCS_P(a)$. If $\mathcal{W}_P(a) \neq \emptyset$, then by the definition of $LCS_P(a)$, there must exist a witness $\langle w_1, w_2\rangle$ such that $w_1 = m - a - LCS_P(a)$. Since $m - b > LCP_P(a)$ and $P[1 : m - b] \approx P[b - a + 1 : m - a]$ by Lemma 4.11, $\langle w_1, w_2\rangle \ominus (b - a) \in \mathcal{W}_P(b)$. ■

Theorem 4.39. *A witness table for OPPM can be constructed in in $O(\log m)$ time and $O(m \log m)$ work on the P-CRCW PRAM.*

Pattern searching

Since the dueling can be performed in $O(1)$ work om P-CRCW PRAM, given a witness location pair, the dueling stage can be performed in $O(\log^3 m)$ time and $O(n \log^2 m)$ work on the P-CRCW PRAM.

Now, let us consider the sweeping stage for OPPM. Suppose that $T_x \not\approx P$ and let $\langle w_1, w_2 \rangle$ be a tight mismatch position pair. For candidates T_{x+a} that are consistent with T_x, if $a + 1 \leq w_1$ and $w_2 \leq a + m$, then $(x + a)$ cannot be a pattern occurrence.

Recall that for each stage k, C is divided into 2^k-blocks. For a 2^k-block, the sweeping stage algorithm (Algorithm 19) chooses x such that x is the smallest index in the second half of the 2^k-block such that $C[x] = \mathit{True}$. For the b-th 2^k-block, we denoted such position x as $x_{b,k}$. In Algorithm 19, we have introduced arrays $\mathit{Cand}[0 : \lfloor m/2^k \rfloor]$ and $\mathit{Mis}[0 : \lfloor m/2^k \rfloor]$, where $\mathit{Cand}[b] = x_{b,k}$ and $\mathit{Mis}[b]$ stored a tight mismatch position pair if $x_{b,k}$ is not a pattern occurrence. If $x_{b,k}$ is a pattern occurrence, $\mathit{Mis}[b] = \langle 0, m + 1 \rangle$.

Next, we discuss how Algorithm 19 ensures that for each round it takes $O(m)$ work. Let R_k be the state of the array R at the beginning of round k. For the b-th 2^k-block of C, let us consider checking if $x_{b,k}$ is a pattern occurrence or not. If we suppose that $\langle w_1, w_2 \rangle$ is a tight mismatch position pair for $T_{x_{b,k}} \not\approx P$, then $T_{x_{b,k}}[1 : w_2 - 1] \approx P[1 : w_2 - 1]$. Thus, for $a > 0$, $T_{x_{b,k}+a}[1 : w_2 - 1 - a] \approx P[1 : w_2 - 1 - a]$. Algorithm 19 updates $R[x_{b,k} + a]$ to $w_2 - 1 - a$. For the case when $T_{x_{b,k}} \approx P$, the discussions above also hold, if we substitute $m + 1$ into w_2.

Theorem 4.40. *The pattern searching for OPPM runs in $O(\log^3 m)$ time and $O(n \log^2 m)$ work on the P-CRCW PRAM.*

Chapter 5

Conclusion and Future Work

In this dissertation, we proposed new serial and parallel algorithms based on the dueling technique [44] for various pattern matching problems. We propose serial and parallel duel-and-sweep algorithms that are generalized for substring consistent equivalence relations (SCER) and consider specific instances of SCER that include parameterized matching, order-preserving matching, palindrome matching and cartesian tree matching.

In Chapter 3, we discuss our serial algorithms for SCER. For SCER, we have proposed an algorithm that uses stack to keep account of the consistent candidates in the dueling stage. In the sweeping stage, verifying surviving candidates from left to right, while taking advantage of the fact that the remaining candidates are pairwise consistent, enabled us to "sweep" through them efficiently. For the pattern preprocessing, we have given algorithm for finding $LCP_P(i)$ for all $0 \leq i < m$. After obtaining this information, a witness table can be easily built from it. It is of independent interest to investigate correlation between border arrays and witness tables for SCER. We will leave it as our future work.

In the second half of Chapter 3, we have discussed specific instances of SCER serial algorithm. We have shown that parameterized matching, cartesian-tree, palindrome and order-preserving matchings can be solved in $O(n)$ time. The preprocessing time for palindrome and cartesian-tree matching is $O(m)$, while for palindrome and order-preserving matchings the preprocessing takes $O(m \log |\Pi|)$ and $O(m \log m)$ time, respectively. These

results are as fast as the existing KMP-based algorithms for these matchings. For the order-preserving matching, instead of relying on the encoding, we focused on the pairs of elements in the string and their relative orders in the pair. Our serial order-preserving algorithm encodes only the pattern, once, and does not require re-encodings of substrings. For our two-dimensional serial algorithm for order-preserving matching, we have reduced the problem of two-dimensional pattern searching into the problem of one-dimensional pattern searching by traversing substrings of the text and the pattern from left-to-right/top-to-bottom fashion. Our algorithm improves the naive 2d algorithm by a factor of m for both pattern preprocessing and pattern searching.

In Chapter 4, we have discussed our parallel algorithm for SCER. Parallelizing SCER pattern matching algorithm revealed interesting properties regarding periodicity of SCER. Vishkin's parallel algorithm for exact matching [44] largely benefited from the fact that, for the exact matching, the periodicity lemma holds and all block-based periods are border-based periods. For SCER, neither of them hold. We have proposed a different approach to solve the pattern matching problem for SCER. Our algorithm is the first parallel algorithm to solve SCER pattern matching problem in parallel.

Given a witness table, $\widetilde{P}$, and $\widetilde{T}$, we have shown that pattern searching under any SCER can be performed in $O(\xi^{\mathrm{t}}_m \log^3 m)$ time and $O(\xi^{\mathrm{w}}_m n \log^2 m)$ work on P-CRCW PRAM. Given $\widetilde{P}$, a witness table can be constructed in $O(\xi^{\mathrm{t}}_m \log^2 m)$ time and $O(\xi^{\mathrm{w}}_m \cdot m \log^2 m)$ work on P-CRCW PRAM. Also, for the case when the pattern is aperiodic, we have shown that the SCER pattern matching can be solved in logarithmic time and linear work complexities, if the re-encoding can be performed in constant time and work. The third condition of $\approx$-encoding in Definition 2.3 ensures the generality of our duel-and-sweep algorithm for SCERs. Although, some encoding methods like the nearest neighbor encoding for order-preserving matching does not fulfill the third condition, Jargalsaikhan et al. [33, 34] showed that there exists an algorithm that is similar to the SCER algorithm proposed in this paper, using the nearest neighbor encoding. In our

future work, we would like to investigate into the relation between the encoding function and the dueling technique and further generalize the definition of encoding so that it becomes more inclusive.

In the second half of Chapter 4, we have discussed parallel algorithms for parameterized, cartesian-tree, palindrome and order-preserving matchings. We have given optimizations for pattern preprocessing, which improves on the SCER extensions by a factor of $\log m$. These optimizations are based on the following fact. Given strings X and Y of equal lengths, $X \approx Y \Leftrightarrow X^R \approx Y^R$. We call such SCER reversible SCER. Parameterized, palindrome and order-preserving matchings fall under reversible SCER. We also show how to compute encodings for above matchings in parallel. Our algorithms are the first algorithm to solve pattern matching problems for these matchings in parallel.

References

[1] Amihood Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.

[2] Amihood Amir, Gary Benson, and Martin Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM Journal on Computing*, 23(2):313–323, 1994.

[3] Amihood Amir and Eitan Kondratovsky. Sufficient conditions for efficient indexing under different matchings. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[4] Alberto Apostolico, Dany Breslauer, and Zvi Galil. Parallel detection of all palindromes in a string. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 497–506. Springer, 1994.

[5] Brenda S Baker. Parameterized pattern matching: Algorithms and applications. *Journal of computer and system sciences*, 52(1):28–42, 1996.

[6] Brenda S Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

[7] Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.

[8] Allan Borodin and John E Hopcroft. Routing, merging, and sorting on parallel models of computation. *Journal of computer and system sciences*, 30(1):130–145, 1985.

[9] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[10] Richard P Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974.

[11] Dany Breslauer and Zvi Galil. An optimal $\mathcal{O}(\log \log n)$ time parallel string matching algorithm. *SIAM Journal on Computing*, 19(6):1051–1058, 1990.

[12] Ayelet Butman, Revital Eres, and Gad M Landau. Scaled and permuted string matching. *Information processing letters*, 92(6):293–297, 2004.

[13] Domenico Cantone, Simone Faro, and M. Oguzhan Külekci. An efficient skip-search approach to the order-preserving pattern matching problem. In *PSC*, pages 22–35, 2015.

[14] Tamanna Chhabra, M. Oguzhan Külekci, and Jorma Tarhio. Alternative algorithms for order-preserving matching. In *PSC*, pages 36–46, 2015.

[15] Tamanna Chhabra and Jorma Tarhio. A filtration method for order-preserving matching. *Information Processing Letters*, 116(2):71–74, 2016.

[16] Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Information Processing Letters*, 115(2):397–402, 2015.

[17] Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In *Stringology*, pages 105–117, 2009.

[18] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[19] Richard Cole, Carmit Hazay, Moshe Lewenstein, and Dekel Tsur. Two-dimensional parameterized matching. *ACM Transactions on Algorithms (TALG)*, 11(2):12, 2014.

[20] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[21] Richard Cole and Uzi Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and computation*, 81(3):334–352, 1989.

[22] Stephen A Cook. The classification of problems which have fast parallel algorithms. In *International Conference on Fundamentals of Computation Theory*, pages 78–93. Springer, 1983.

[23] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016. Pattern Matching, Text Data Structures and Compression.

[24] Christos Faloutsos. *Searching multimedia databases by content*, volume 3. Springer Science & Business Media, 2012.

[25] Simone Faro and M. Oğuzhan Külekci. Efficient algorithms for the order preserving pattern matching problem. In *International Conference on Algorithmic Applications in Management*, pages 185–196. Springer, 2016.

[26] Nathan J Fine and Herbert S Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965.

[27] Zvi Galil. Optimal parallel algorithms for string matching. *Information and Control*, 67(1-3):144–157, 1985.

[28] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[29] Md Mahbubul Hasan, ASM Shohidull Islam, Mohammad Saifur Rahman, and M Sohel Rahman. Order preserving pattern matching revisited. *Pattern Recognition Letters*, 55:15–21, 2015.

[30] Xin He and Yaacov Yesha. Binary tree algebraic computation and parallel algorithms for simple graphs. *Journal of Algorithms*, 9(1):92–113, 1988.

[31] R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[32] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.

[33] Davaajav Jargalsaikhan, Diptarama, Yohei Ueki, Ryo Yoshinaka, and Ayumi Shinohara. Duel and sweep algorithm for order-preserving pattern matching. In *SOFSEM 2018: Theory and Practice of Computer Science*, pages 624–635, 2018.

[34] Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Parallel duel-and-sweep algorithm for the order-preserving pattern matching. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 211–222. Springer, 2020.

[35] Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.

[36] Donald E. Knuth, James H. Morris, Jr, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[37] Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

[38] Glenn Manacher. A new linear-time"on-line"algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM (JACM)*, 22(3):346–351, 1975.

[39] Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theoretical Computer Science*, 656:225–233, 2016.

[40] Gary L Miller and John H Reif. Parallel tree contraction and its application. Technical report, HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB, 1985.

[41] Sung Gwan Park, Amihood Amir, Gad M. Landau, and Kunsoo Park. Cartesian tree matching and indexing. In *30th Annual Symposium on Combinatorial Pattern Matching*, pages 16:1–16:14, 2019.

[42] I Tomohiro, Shunsuke Inenaga, and Masayuki Takeda. Palindrome pattern matching. In *Annual Symposium on Combinatorial Pattern Matching*, pages 232–245. Springer, 2011.

[43] Yohei Ueki, Kazuyuki Narisawa, and Ayumi Shinohara. A fast order-preserving matching with q-neighborhood filtration using SIMD instructions. In *SOFSEM (Student Research Forum Papers/Posters)*, pages 108–115, 2016.

[44] Uzi Vishkin. Optimal parallel pattern matching in strings. In *International Colloquium on Automata, Languages, and Programming*, pages 497–508. Springer, 1985.

[45] Uzi Vishkin. Deterministic sampling — a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, 1991.

List of Publications

Conference papers

1. Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Parallel duel-and-sweep algorithm for the order-preserving pattern matching. In *the International Conference on Current Trends in Theory and Practice of Informatics*, 211-222. 2020.

2. Ryu Wakimoto, Satoshi Kobayashi, Yuki Igarashi, Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. AOBA: An Online Benchmark Tool for Algorithms in Stringology. In *the International Conference on Current Trends in Theory and Practice of Informatics (Doctoral Student Research Forum)*, 1-12. 2020.

3. Davaajav Jargalsaikhan, Diptarama, Yohei Ueki, Ryo Yoshinaka, and Ayumi Shinohara. Duel and sweep algorithm for order-preserving pattern matching. In *the International Conference on Current Trends in Theory and Practice of Informatics*, 624–635. 2018.

4. Davaajav Jargalsaikhan, Naoaki Okazaki, Koji Matsuda, and Kentaro Inui. Building a corpus for japanese Wikification with fine-grained entity classes. In *the Proceedings of the ACL 2016 Student Research Workshop*, 138–144. 2016.