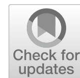
Acta Informatica
https://doi.org/10.1007/s00236-024-00464-w

ORIGINAL ARTICLE



Serial and parallel algorithms for order-preserving pattern matching based on the duel-and-sweep paradigm

Davaajav Jargalsaikhan[1] · Diptarama Hendrian[1] · Yohei Ueki[1] · Ryo Yoshinaka[1] · Ayumi Shinohara[1]

Received: 1 April 2024 / Accepted: 9 August 2024
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2024

Abstract
Given a text and a pattern over an alphabet, the classic exact matching problem searches for all occurrences of the pattern in the text. Unlike exact matching, *order-preserving pattern matching* (OPPM) considers the relative order of elements, rather than their exact values. In this paper, we propose efficient algorithms for the OPPM problem using the "duel-and-sweep" paradigm. For a pattern of length m and a text of length n, our serial algorithm runs in $O(n+m \log m)$ time, and our parallel algorithm runs in $O(\log^2 m)$ time and $O(n \log^2 m)$ work with $O(\log m)$ time and $O(m \log m)$ work pattern preprocessing on the Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM).

1 Introduction

The exact string matching problem is a widely studied problem in the field of computer science. Given a text and a pattern, the exact matching problem searches for all occurrence positions of the pattern in the text. Many pattern matching algorithms have been proposed such as the well-known Knuth-Morris-Pratt algorithm [22], Boyer-Moore algorithm [4], Horspool algorithm [15]. Faro and Lecroq [13] summarizes recent results on pattern matching algorithms. Previously proposed pattern matching algorithms preprocess the pattern first and then match the pattern from its prefix or suffix when comparing it with the text. Vishkin

Davaajav Jargalsaikhan, Diptarama Hendrian, Yohei Ueki, Ryo Yoshinaka and Ayumi Shinohara have contributed equally to this work.

✉ Diptarama Hendrian
diptarama.hendrian@tmd.ac.jp

✉ Ryo Yoshinaka
ryoshinaka@tohoku.ac.jp

✉ Ayumi Shinohara
ayumis@tohoku.ac.jp

Davaajav Jargalsaikhan
davaajav_jargalsaikhan@shino.ecei.tohoku.ac.jp

Yohei Ueki
yohei_ueki@shino.ecei.tohoku.ac.jp

1 Graduate School of information Sciences, Tohoku University, Aobaku-aza-Aoba, Sendai 980-8579, Miyagi, Japan

proposed two algorithms for pattern matching, pattern matching by duel-and-sweep [26] and pattern matching by sampling [27]. 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.

Furthermore, variants of Vishkin's duel-and-sweep algorithm have been developed for other types of pattern matching. Amir et al. [2] proposed a duel-and-sweep algorithm for the two-dimensional pattern matching problem. Cole et al. [10] generalized it for two-dimensional parameterized pattern matching. Note that those algorithms are serial ones, whereas Vishkin's original algorithms are parallel. Recently, Jargalsaikhan et al. [19] proposed a general parallel algorithm for a family of pattern matching problems based on Vishkin's duel-and-sweep paradigm (see also [20] for an improvement). Namely, their proposed algorithm solves pattern matching problems under arbitrary *substring consistent equivalent relations (SCERs)*. A matching equivalent relation is said to be SCER if whenever two strings match, they have the same length and every pair of their substrings at the same position also matches. Representative SCERs include exact matching, parameterized matching, Cartesian tree matching, and *order-preserving matching (OPM)*. The efficiency of the parallel SCER matching algorithm depends on encodings of strings under SCERs that in some sense reduce SCERs in concern to exact matching. Indeed, Amir and Kondratovsky [1] showed that every SCER admits such an encoding. While the encoding given in the proof of their general theorem is computationally expensive, the standard and cheap encodings used in parameterized matching and Cartesian tree matching satisfy the requirement to be used in the algorithm. However, the standard encoding for OPM does not meet the requirement (see Appendix A). Therefore, "efficient" duel-and-sweep parallel algorithms for solving the order-preserving pattern matching problem (OPPMP) are not yet known.[1]

Unlike the exact matching problem, the OPPMP considers the relative order of elements, rather than their exact values. For instance, (12, 35, 5) and (25, 30, 21) do not match in the exact matching sense. However, for OPM, (12, 35, 5) is considered to match (25, 30, 21), since their relative orders of the elements coincide. Namely, the first element is the median, the second element is the largest, and the third element is the smallest among (12, 35, 5) and (25, 30, 21), respectively. The OPPMP 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. The difficulty of the OPPMP 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. In OPM, the two concatenations will not necessarily match each other. For instance, (12, 35, 5) and (25, 30, 21) match, but their two concatenations (12, 35, 5, 25, 30, 21) and (25, 30, 21, 12, 35, 5) do not.

Kubica et al. [23] and Kim et al. [21] independently proposed the same solution for the OPPMP based on the KMP algorithm. Their KMP-based algorithm runs in $O(n + m \log m)$ time. Cho et al. [8] brought forward another algorithm based on the Horspool algorithm that uses q-grams, which was proven to be experimentally fast. Crochemore et al. [11] proposed useful data structures for the OPPMP. On the other hand, Chhabra and Tarhio [6], Faro and Külekci [12] proposed filtration methods which are practically fast. Moreover, faster filtration algorithms using SIMD (Single Instruction Multiple Data) instructions were

[1] Our preliminary paper [18] on this topic presented in SOFSEM 2020 is in error.

proposed by Cantone et al. [5], Chhabra et al. [7] and Ueki et al. [25]. They showed that SIMD instructions are effective in speeding up their algorithms.

In this paper, we propose new serial and parallel algorithms for the OPPMP based on the duel-and-sweep technique. Given the text of length n and the pattern of length m, our serial algorithm runs in $O(n + m \log m)$ time which is as fast as the KMP based algorithm. Our parallel algorithm runs in $O(\log^2 m)$ time using $O(n \log^2 m)$ work on the Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM) [16]. 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. 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. To the best of our knowledge, our parallel algorithm is the first efficient one to solve the OPPMP. Our parallel algorithm is based on the one for general SCERs by Jargalsaikhan [19, 20]. Our proposal does not only evade the costs of encoding used in the general algorithm, but is tuned using a special property of OPM that general SCERs do not necessarily satisfy.

The rest of this work is organized as follows. In Sect. 2, we describe the notation and give definitions that we will use for our algorithms. In Sect. 3, we describe the idea of duel-and-sweep algorithm and discuss our serial algorithm. In Sect. 4, we give a parallel algorithm for computing the encoding for order-preserving matching and describe our parallel algorithm. Lastly, we conclude our work in Sect. 5. In Appendix, we compare our parallel OPM algorithm and the one for general SCERs proposed in [19].

Preliminary versions of this paper appeared in [17, 18]. However, the pattern preprocessing algorithm in the latter is in error and fixed in this paper.

2 Preliminaries

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 the string of length 0. Throughout this paper, strings are 1-indexed, unless otherwise stated. For a string $X \in \Sigma^*$, we will denote the i-th element of X by $X[i]$ and the substring of X that starts at the position i and ends at j as $X[i : j] = X[i]X[i+1] \dots X[j]$. For convenience, we abbreviate $X[1 : i]$ to $X[: i]$ and $X[i : |X|]$ to $X[i :]$, which are called a *prefix* and a *suffix* of X, respectively. Moreover, let $X[i : j] = \varepsilon$ if $i > j$. In addition, we denote the subsequence $X[i]X[j]$ of X constituted by $X[i]$ and $X[j]$ as $X[\langle i, j \rangle]$ or as $X[i, j]$.

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

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

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 such that the condition above does not hold. We will call such $\langle i, j \rangle$ with $i < j$ a *mismatch position pair* for X and Y. In other words, $\langle i, j \rangle$ is a mismatch position pair iff $X[\langle i, j \rangle] \not\approx Y[\langle i, j \rangle]$. We say that a mismatch position pair $\langle i, j \rangle$ is *prefix-tight* if $X[1 : j-1] \approx Y[1 : j-1]$ and $X[1 : j] \not\approx Y[1 : j]$. Symmetrically, $\langle i, j \rangle$ is called *suffix-tight* if $X[i+1 : n] \approx Y[i+1 : n]$ and $X[i : n] \not\approx Y[i : n]$. For instance, concerning $(25, 30, 21, 18) \not\approx (11, 13, 20, 15)$, we have several mismatch position pairs. Among those,

$\langle 1, 3\rangle$ is prefix-tight, $\langle 2, 3\rangle$ is prefix-tight and suffix-tight, and $\langle 1, 4\rangle$ is neither prefix-tight nor suffix-tight, for example. Because this paper uses prefix-tight mismatch position pairs much more often than suffix-tight ones, we simply call the former *tight*. For the rest of the paper, we define binary operations $\oplus$ and $\ominus$ for shifting a position pair by an offset. 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 $\min\langle i, j\rangle = \min\{i, j\}$.

Suppose that we are given a text T of length n and a pattern P of length m. We call every integer x with $1 \leq x \leq n-m+1$ a *candidate position*, and the substring $T_x = T[x:x+m-1]$ of T starting from x of length m a *candidate*. If no confusion arises, by "candidate" we also mean "candidate position". When a candidate T_x is order-isomorphic to the pattern P, we call its position x an *occurrence* of the pattern P inside the text T. The *order-preserving pattern matching problem* (OPPM problem) is defined as follows.

Definition 1 (Order-preserving pattern matching)

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

In order to check the order-isomorphism of a string X with another string, Kubica et al. [23] defined useful arrays $Lmax_X$ and $Lmin_X$ so that $Lmax_X[i]$ and $Lmin_X[i]$ are the positions j and k left to i such that $X[j]$ and $X[k]$ are next largest and smallest to $X[i]$, respectively. If there is a tie, we pick the rightmost position. If $X[i]$ is strictly smaller than any of $X[1], \ldots, X[i-1]$, then $Lmax_X[i] = 0$. Similarly if $X[i]$ is strictly larger than any of $X[1], \ldots, X[i-1]$, then $Lmin_X[i] = 0$. More formally,

$$Lmax_X[i] = \begin{cases} \max\{\, j < i \mid X[j] = \max S_i \,\} & \text{if } S_i \neq \emptyset, \\ 0 & \text{otherwise,} \end{cases}$$

$$\text{where } S_i = \{\, X[j] \mid 1 \leq j < i \text{ and } X[j] \leq X[i] \,\},$$

$$Lmin_X[i] = \begin{cases} \max\{\, k < i \mid X[k] = \min L_i \,\} & \text{if } L_i \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

$$\text{where } L_i = \{\, X[k] \mid 1 \leq k < i \text{ and } X[k] \geq X[i] \,\}.$$

An example is shown in Table 1. Clearly, for a position $i \in \{1, \ldots, m\}$ such that $Lmax_X[i] \neq 0$ and $Lmin_X[i] \neq 0$,

$$X[Lmax_X[i]] = X[i] \iff X[i] = X[Lmin_X[i]],$$
$$X[Lmax_X[i]] < X[i] \iff X[i] < X[Lmin_X[i]].$$

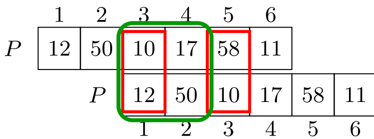
We can easily observe that $X \approx Y$ iff $Lmax_X = Lmax_Y$ and $Lmin_X = Lmin_Y$. However, we can decide order-isomorphism between X and Y referring to $Lmax_X$, $Lmin_X$, and Y, without computing $Lmax_Y$ and $Lmin_Y$, based on Lemma 2 below. We first define $F_X(Y, i)$ for $Y \in \Sigma^m$ and $1 \leq i \leq m$ by

$$F_X(Y, i) = \begin{cases} i_{\max} & \text{if } i_{\max} \neq 0 \text{ and } Y[i_{\max}] > Y[i] \text{ for } i_{\max} = Lmax_X[i], \\ i_{\min} & \text{if } i_{\min} \neq 0 \text{ and } Y[i_{\min}] < Y[i] \text{ for } i_{\min} = Lmin_X[i], \\ 0 & \text{otherwise.} \end{cases} \quad (1)$$

If both conditions in Eq. (1) hold, either $i_{\max}$ or $i_{\min}$ can be taken. Provided that $Lmax_X$ and $Lmin_X$ are prepared, one can compute $F_X(Y, i)$ in constant time.

Table 1 The $Lmax_X$ and $Lmin_X$-arrays for $X = (12, 50, 10, 17, 58, 11)$

	1	2	3	4	5	6
X	12	50	10	17	58	11
$Lmax_X$	0	1	0	1	2	3
$Lmin_X$	0	0	1	2	0	1



	1	2	3	4	5
LIP_P	1	2	3	1	1
W	$\langle 1, 2\rangle$	$\langle 1, 3\rangle$	$\langle 0, 0\rangle$	$\langle 1, 2\rangle$	$\langle 0, 0\rangle$

Fig. 1 Suppose $P = (12, 50, 10, 17, 58, 11)$ is superimposed on itself with offset 2. Then, we see $P[3:4] \approx P[1:2]$ but $P[3:5] \not\approx P[1:3]$, by $P[3] \leq P[5]$ and $P[1] > P[3]$. This gives $LIP_P(2) = 2$. The position pair $\langle 1, 3\rangle$ is called a witness for offset 2. The length of the longest isomorphic prefixes and witnesses for other offsets are summarized in the right table, where $W[i] = \langle 0, 0\rangle$ means that offset i has no witness, i.e., i is a period

Lemma 2 [8] For two strings X and Y of length m, assume that $X[1 : i-1] \approx Y[1 : i-1]$ for some $0 < i \leq m$. Then $X[1 : i] \approx Y[1 : i]$ iff $F_X(Y, i) = 0$.

Therefore, $X \approx Y$ if and only if $F_X(Y, i) = 0$ for all $i \leq m$. In the case where $X \not\approx Y$, the function F_X gives an evidence.

Lemma 3 *For two strings X and Y of length m, if $j = F_X(Y, i) \neq 0$ for some $1 \leq i \leq m$, then $\langle j, i\rangle$ is a mismatch position pair for $X \not\approx Y$.*

Proof Since $j \neq 0$, either $j = Lmin_X[i]$ and $Y[j] < Y[i]$, or $j = Lmax_X[i]$ and $Y[j] > Y[i]$. In the former case, since $X[j] \geq X[i]$ by the definition of $Lmin_X[i]$, $\langle j, i\rangle$ is a mismatch position pair. For the latter case, since $X[j] \leq X[i]$ by the definition of $Lmax_X[i]$, $\langle j, i\rangle$ is a mismatch position pair. □

The function F_X can work for comparing Y shorter than X against the prefix of X of length $|Y| < |X|$, since $Lmax_X[:j] = Lmax_{X[:j]}$ and $Lmin_X[:j] = Lmin_{X[:j]}$ for $j = |Y| < |X|$.

By $LIP(X, Y)$, we denote the length of the longest isomorphic prefixes of two strings X and Y of the same length m. That is, $LIP(X, Y)$ is the largest integer $\ell \leq m$ such that $X[:\ell] \approx Y[:\ell]$. Furthermore, for a single string X, we define the *LIP*-function LIP_X, which is essentially identical to the *Z-array* [14]. For an integer $0 \leq a < m$, we define $LIP_X(a) = LIP(X[1 : m-a], X[a+1 : m])$. In other words, $LIP_X(a)$ is the length of the longest isomorphic prefixes, when X is superimposed on itself with offset a. Obviously, $LIP_X(a) = \ell < m-a$ iff there exists $i \leq \ell$ such that $\langle i, \ell+1\rangle$ is a (prefix-)tight mismatching position pair for $X[1 : m-a]$ and $X[a+1 : m]$. Fig. 1 shows an example.

Symmetrically, $LIS_X(a)$ denotes the length of the longest isomorphic suffixes, when X is superimposed on itself with offset a. That is, given an integer $0 \leq a < m = |X|$, $LIS_X(a) = \ell$ is the greatest integer such that $X[m-\ell+1 : m] \approx X[m-a-\ell+1 : m-a]$. We have $LIS_X(a) = \ell < m-a$ iff there exists $j > m-a-\ell$ such that $\langle m-a-\ell,\ j\rangle$ is a suffix-tight mismatching position pair for $X[1 : m-a]$ and $X[a+1 : m]$.

Let $\text{rev}(X)$ be the reverse of X, which can be inductively defined by $\text{rev}(\varepsilon) = \varepsilon$ and $\text{rev}(AX) = \text{rev}(X)A$ for any $A \in \Sigma$ and $X \in \Sigma^*$. Since, $X \approx Y \Leftrightarrow \text{rev}(X) \approx \text{rev}(Y)$ for any two strings X and Y, $LIS_X(a) = LIP_{\text{rev}(X)}(a)$ for any offset a. Throughout this paper, an offset is always a non-negative integer.

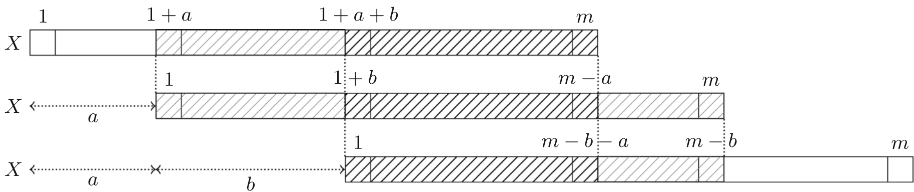


Fig. 2 Illustration to Lemma 5. The vertically aligned shaded regions are mutually order-isomorphic

Vishkin's dueling technique essentially depends on the preferable properties of *periods* of strings. Matsuoka et al. [24] have discussed in detail how the classical notion of periods and their properties can be generalized when considering SCER matching. Unfortunately, none of the generalizations yield a straightforward adaptation of Vishkin's algorithm for order-preserving matching. Among those, the kind of periods involved in the duel-and-sweep algorithm discussed in this paper is *border-based period*.

Definition 4 (Border-based period) Given a string X of length m, positive integer $p < m$ is called a *border-based period* of X if $X[1 : m - p] \approx X[p + 1 : m]$.

Throughout the rest of the paper, we will refer to a border-based period as a *period*. By definition, $p < m$ is a period of X iff $LIP_X(p) = m - p$. The string $P = (12, 50, 10, 17, 58, 11)$ in Fig. 1 has periods 3 and 5.

Lemma 5 *If a and b are periods of X and $a + b < |X|$, then $(a + b)$ is a period of X.*

Proof See Fig. 2. Let $m = |X|$. Since a is a period of X, by definition $X[1 : m - a] \approx X[1 + a : m]$. Since suffixes of order-isomorphic strings of the same length are also order-isomorphic, $X[1 + b : m - a] \approx X[1 + a + b : m]$. Similarly, since b is a period of X, $X[1 : m - b] \approx X[1 + b : m]$, and thus $X[1 : m - b - a] \approx X[1 + b : m - a]$. Hence, $X[a + b + 1 : m] \approx X[1 : m - b - a]$, which means that $(b + a)$ is a period of X. □

3 Serial duel-and-sweep algorithm for OPPM

In this section we describe our serial algorithm for OPPM. Before discussing our algorithm in detail, we give an overview of the "duel-and-sweep" paradigm [2, 26], which is applicable to both our serial and parallel algorithms. In the remainder of this paper, we fix text T to be of length n and pattern P to be of length m.

3.1 Overview of the duel-and-sweep algorithm

In the duel-and-sweep paradigm, candidates are pruned in two stages, called the *dueling* and the *sweeping* stages. Suppose P is superimposed on itself with an offset $a < m$. If a is not a period of P, the two overlapped regions of P, i.e., $P[1 : m - a]$ and $P[1 + a : m]$, are not order-isomorphic. Then, it is impossible for two candidates with offset a to be both order-isomorphic to 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 $T_x \approx P$. On the other hand, if candidates T_x

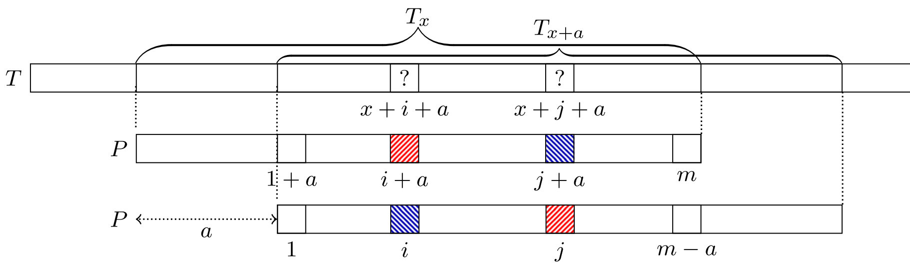


Fig. 3 Duel between T_x and T_{x+a}. Assume that a is not a period of P and that $P[1:m-a]$ and $P[1+a:m]$ have a mismatch position pair $w = \langle i, j \rangle$, i.e., $P[w] \not\approx P[w \oplus a]$. Then, by comparing $T[w \oplus (x+a)] = T_x[w \oplus a] = T_{x+a}[w]$ with $P[w]$, one can eliminate one of T_x and T_{x+a}. If $T[w \oplus (x+a)] \approx P[w]$, then $T_x \not\approx P$. If $T[w \oplus (x+a)] \not\approx P[w]$, then $T_{x+a} \not\approx P$.A concrete example can be found in Fig. 4

and T_{x+a} are overlapping, i.e. $a < m$, and its offset a is a period of P, then they do not duel and both of them may survive the dueling stage. Then, the suffixes of T_x and P of length $m-a$ match if and only if so do the prefixes of T_{x+a} and P of the same length. The sweeping stage takes the advantage of this property when checking the order-isomorphism between surviving candidates and the pattern so that this stage can be done also quickly.

For a non-period offset $a < m$, where the overlapped regions obtained by superimposing P on itself with offset a do not match, the original duel-and-sweep algorithm [26] for exact matching saves a single position i such that $P[i] \neq P[i+a]$. Such position i is called a *witness for the offset a*. However, in OPM, order-isomorphism of two strings cannot be refuted by comparing a symbol in one position. One way to overcome this difficulty is to transform the pattern and candidates by appropriate encoding so that comparing the symbols at a single position is sufficient. This is what Jargalsaikhan et al. [19] did in their parallel algorithm for general SCER matching. This technique is applicable to OPM in principle, but actually, no such computationally cheap encoding is known for OPM (See Appendix A). Instead, we use two positions as a witness to say that the two strings are not order-isomorphic. When the overlapped regions obtained by superimposing P on itself with offset a are not order-isomorphic, i.e., $P[:m-a] \not\approx P[1+a:]$, there is a position pair $\langle i, j \rangle$ such that $P[:m-a][i,j] = P[i,j] \not\approx P[i+a, j+a] = P[1+a:][i,j]$, which we call a *witness (pair) for offset a* (Fig. 1); That is, either

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

For the rest of this paper, we assume $i < j$ for any witness pair $\langle i, j \rangle$. We denote by $\mathcal{W}_P(a)$ the set of all witnesses for offset a:

$$\mathcal{W}_P(a) = \{ \langle i, j \rangle \mid P[i,j] \not\approx P[i+a, j+a] \text{ and } 1 \leq i < j \leq m-a \}$$

Obviously, $\mathcal{W}_P(a) = \emptyset$ iff a is a period of P or $a = 0$.

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. A *witness table* $W[1:m-1]$ is an array such that $W[a] \in \mathcal{W}_P(a)$ unless $\mathcal{W}_P(a) = \emptyset$. When $\mathcal{W}_P(a) = \emptyset$, which means a is a period, we express it as $W[a] = \langle 0, 0 \rangle$. Hereinafter, we will refer to $\langle 0, 0 \rangle$ as a *zero*. Figure 1 shows an example of a witness table.

The dueling stage "duels" pairs of candidates T_x and T_{x+a} for non-periods a, i.e., $W[a] \neq \langle 0, 0 \rangle$, and we eliminate one of them. Witnesses are used in the following manner. Suppose

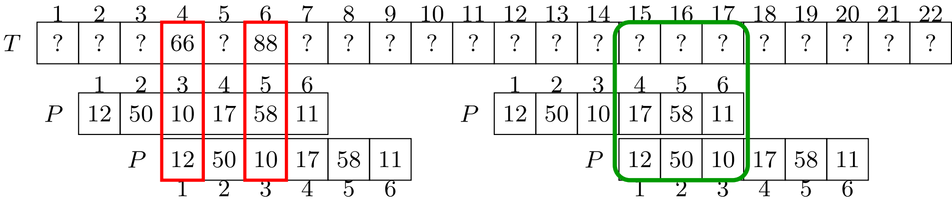


Fig. 4 When $P = (12, 50, 10, 17, 58, 11)$ is superimposed on itself with offset 2, the overlapped regions $P[3{:}6]$ and $P[1{:}4]$ are not order-isomorphic, by $P[3] \leq P[5]$ and $P[1] > P[3]$. Then, for any pair of candidates with offset 2, at least one of them is not order-isomorphic to P. For example, at least one of T_2 and T_4 is not order-isomorphic to P. Since $T[4, 6] = (66, 88) \not\approx (12, 10) = P[1, 3]$, we conclude $T_4 \not\approx P$. The position pair $\langle 1, 3\rangle$ is called a witness for offset 2. On the other hand, when P is superimposed on itself with offset 3, the overlapped regions are order-isomorphic. Candidate positions 12 and 15 are said to be consistent and we do not perform duel between them

that $W[a] = \langle i, j\rangle$, where $P[i] > P[j]$ and $P[i + a] \leq P[j + a]$, for example. Then, it holds that

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

(Figs. 3 and 4). Based on this observation, we can safely eliminate either candidate T_x or T_{x+a} without looking into other positions. We can perform this process similarly for other equality/inequality cases. This process is called *dueling*. On the other hand, if the offset a has no witness pair, i.e. if a is a period of P, no dueling is performed on them. We say that a position x is *consistent with* $x + a$ if a is a period of P or $a \geq m$. The consistency property is transitive.

Lemma 6 *For any* x, y, z *such that* $0 < x < y < z < n$, *if* x *is consistent with* y *and* y *is consistent with* z, *then* x *is consistent with* z.

Proof If $z - x \geq m$, we have nothing to prove. Suppose $z - x < m$. By Lemma 5, if $(y - x)$ and $(z - y)$ are periods, then so is $(y - x) + (z - y) = (z - x)$. □

After the dueling stage, all surviving candidates are pairwise consistent. Taking advantage of this property, the sweeping stage prunes the surviving candidates until all remaining candidates are order-isomorphic to the pattern. In other words, the sweeping stage finds all occurrences of the pattern inside the text.

3.2 Pattern preprocessing

The goal of the preprocessing stage, described in Algorithm 1, is to compute a witness table $W[1 : m - 1]$. First, we construct the arrays $Lmax_P$ and $Lmin_P$. In addition, we construct the *Z-array* Z_P, which is defined by $Z_P[a] = LIP_P(a - 1)$ for $1 \leq a \leq m$.

Lemma 7 [23] For a string X of length m, $Lmax_X$ and $Lmin_X$ can be computed in $O(m \log m)$ time.

Lemma 8 [14] Given that $Lmax_X$ and $Lmin_X$ are already computed for a string X of length m, Z_X can be computed in $O(m)$ time.

Using the value of $LIP_P(a) = \ell$, we can verify whether $\mathcal{W}_P(a)$ is empty or not. If $\ell = m - a$, a is a period of P and thus $\mathcal{W}_P(a) = \emptyset$. If $\ell < m - a$, then $\mathcal{W}_P(a) \neq \emptyset$ and there must exist a position pair $\langle i, \ell + 1\rangle \in \mathcal{W}_P(a)$ for some $i \leq \ell$.

Algorithm 1: Serial algorithm for the pattern preprocessing

```
1  Function PreprocessingSerial(P)
2      create array of integer pairs W[0 : m − 1];
3      compute arrays Lmin_P, Lmax_P, and Z_P;
4      for a = 0 to m − 1 do
5          if Z_P[a + 1] = m − a then
6              W[a] ← ⟨0, 0⟩;
7          else
8              j ← Z_P[a + 1] + 1;
9              i ← F_P(P[a + 1 : ], j);            /* Lmin_P and Lmax_P are used here */
10             W[a] ← ⟨i, j⟩;
11     return W;
```

Algorithm 2: Serial algorithm for the dueling stage

```
1  Function DuelingStageSerial(P, T, 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 y − x ≥ m or W[y − x] = ⟨0, 0⟩ then
7                  push x and y to stack;
8                  break;
9              else
10                 ⟨i, j⟩ ← W[y − x];
11                 if P[i, j] ≉ T[y + i − 1, y + j − 1] then
12                     push x to stack;
13                     break;
14         if stack is empty then
15             push y to stack;
16     return stack;
```

Lemma 9 *For a pattern P of length m, Algorithm 1 constructs a witness table W in $O(m \log m)$ time.*

Proof Clearly the algorithm runs in $O(m \log m)$ time.

We show that for each $0 \le a < m$, Algorithm 1 computes $W[a]$ correctly. Suppose that $LIP_P(a) = Z_P[a+1] = m-a$. This means that $P[1{:}i-1] \approx P[1+a{:}i-1+a] = P[1+a{:}m]$ for $i = LIP_P(a) + 1$, i.e., there is no witness pair for offset a. Indeed, Algorithm 1 gets $W[a] = \langle 0, 0 \rangle$ for this case.

Suppose that we have $LIP_P(a) = Z_P[a+1] < m - a$, i.e. $P[1 : LIP_P(a)] \approx P[1 + a : LIP_P(a) + a]$ and $P[1 : LIP_P(a) + 1] \not\approx P[1 + a : LIP_P(a) + a + 1]$. Therefore, there must exist a witness for offset a. Let $j = LIP_P(a) + 1$ and $i = F_P(P[1 + a :], j)$. By Lemmas 2 and 3, $i \ne 0$ and $\langle i, j \rangle \in \mathcal{W}_P(a)$. Indeed Algorithm 1 gets $W[a] = \langle i, j \rangle$. □

		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	

Fig. 5 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 $W = (\langle 1, 2\rangle, \langle 0, 0\rangle, \langle 0, 0\rangle)$. 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 defeats T_6 and the contents of the stack become 1, 3, and 7

3.3 Pattern searching

As we have mentioned earlier in this section, the pattern searching consists of the dueling and the sweeping stages. The process of the dueling stage is shown in Algorithm 2. This stage eliminates candidates until all surviving candidates 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 Lemma 6, 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. Figure 5 gives an example run of the dueling stage.

Lemma 10 *The dueling stage can be done in $O(n)$ time by using W.*

In order to check whether some surviving candidate T_x is order-isomorphic to P, it is enough to confirm $F_P(T_x, i) = 0$ for all $1 \le i \le m$ (Eq. 1, Lemma 3). A naive implementation of sweeping requires $O(mn)$ time. Algorithm 3 takes advantage of the fact that all the remaining candidates are pairwise consistent, so that we can reduce the time complexity to $O(n)$. See Figures 6 and 7. Let $j = LIP(T_x, P) + 1$; i.e., $T_x[1 : j-1] \approx P[1 : j-1]$ and $T_x[1 : j] \not\approx P[1 : j]$. This is the smallest integer j such that $F_P(T_x, j) \ne 0$. For the next candidate 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. That is, it is ensured that $F_P(T_{x+a}, i) = 0$ for all $i < j-a$ and thus it suffices to check the values $F_P(T_{x+a}, i) = 0$ for $i \ge j-a$. If $P \approx T_x$, the above discussion holds for $j = m+1$. Therefore, the total number of comparison is bounded by

Algorithm 3: Serial algorithm for the sweeping stage

```
1  Function SweepingStageSerial(P, T, candidate_list)
2      i ← 1;
3      while there are unchecked candidates in candidate_list do
4          let T_x be the leftmost unchecked candidate;
5          start ← max{1, i − x + 1};
6          for j = start to m do
7              if F_P(T_x, j) ≠ 0 then
8                  eliminate T_x;
9                  break;
10             i ← x + j;
```

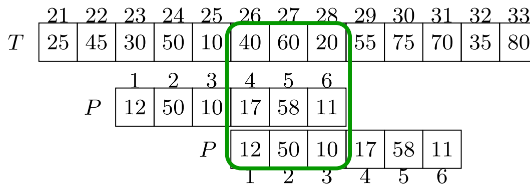


Fig. 6 After the dueling stage, the surviving candidates are pairwise consistent. In this example, T_{23} and T_{26} have survived the dueling stage and are consistent. If we have known that $T_{23} \approx P$, when comparing $T_{26} \approx P$, we can use the fact for free that $T_{26}[1{:}3] = T_{23}[4:6] \approx P[4{:}6] \approx P[1{:}3]$. We start comparison of T_{26} and P from position 4

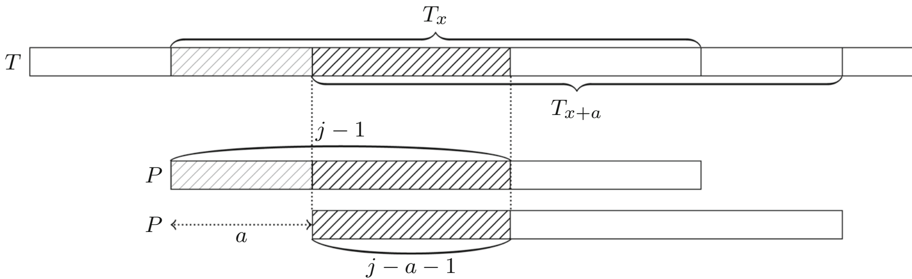


Fig. 7 In the sweeping stage, if $T_x[1:j-1] \approx P[1:j-1]$, it is guaranteed that $T_{x+a}[1:j-a-1] \approx P[1:j-a-1]$ for any period $a < j-1$ of P. So, we can check the isomorphism between T_{x+a} and P from the $(j-a)$th position. A concrete example is found in Figure 6

$O(n)$, by applying the same argument on the complexity of the KMP algorithm for exact matching.

Lemma 11 *The sweeping stage can be completed in $O(n)$ time.*

We conclude this section with the following theorem.

Theorem 12 *Given a text T of length n and a pattern P of length m, the duel-and-sweep algorithm solves the OPPMP in $O(n)$ time with $O(m \log m)$ time preprocessing.*

Proof By Lemmas 9, 10, and 11. □

Algorithm 4: Computes a tight mismatch position pair in parallel

```
1 Function GetMismatchPos(Lmax_X, Lmin_X, Y, r)
2     ⟨w1, w2⟩ ← ⟨0, 0⟩;
3     for each i ∈ {r + 1, ..., |X|} do in parallel
4         j ← F_X(Y, i);                    /* Lmax_X and Lmin_X are used here */
5         if j ≠ 0 then
6             ⟨w1, w2⟩ ⇐ ⟨j, i⟩;
7     return ⟨w1, w2⟩;
```

4 Parallel duel-and-sweep algorithm for OPPM

This section discusses a parallel version of the duel-and-sweep algorithm for OPPM. One easy parallelization is to cut the text into small overlapping pieces and then to run the serial algorithm presented in the previous section for them independently. This idea takes at least $\Omega(m \log m)$ time. Another simple and extreme idea is to use one processor for each position pair (i, j) on the text and then let them compare the values $T[i]$ and $T[j]$ as well as $P[i]$ and $P[j]$ to find mismatches. This idea realizes an algorithm that runs in sublinear time, but requires as much as $\Omega(n^2)$ work. Instead, we in this section will present a more reasonable parallel algorithm, which runs in $O(\log^2 m)$ time and with $(n \log^2 m)$ work. The general framework of the duel-and-sweep algorithm, as we have described in the beginning of Sect. 3, remains the same. To efficiently solve the OPPM problem in parallel, we enrich ideas used in the serial algorithm with new ones. Hereinafter, in our pseudo-codes we will use "$\leftarrow$" to note assignment operation into a local variable of a processor. We will use "$\Leftarrow$" to note assignment operation into a global variable which is accessible from multiple processors simultaneously. In case of a write conflict, the processor with the smallest index succeeds in writing into the memory.

First, we discuss how to compute $Lmax_X$ and $Lmin_X$ in parallel.

Lemma 13 *Given a string X of length m, $Lmax_X$ and $Lmin_X$ can be computed in $O(\log m)$ time and $O(m \log m)$ work on the P-CRCW PRAM.*

Proof Following the construction of $Lmax_X$ and $Lmin_X$ by [23], suppose that positions of X are sorted with respect to their contents. In case of equal contents, the smaller positions come first (stable sort). Let X' be the resulting sequence of positions. For $i \in \{1, \dots, m\}$, let j be the position of i in X', i.e., $X'[j] = X[i]$. Then $Lmax_X[i]$ is the nearest smaller value in X' to the left of $X'[j]$. If there is no such value, $Lmax_X[i] = 0$. $Lmin_X$ is computed similarly. Using the merge sort algorithm by Cole [9] and the all-smaller-nearest-values algorithm by Berkman et al. [3], $Lmax_X$ and $Lmin_X$ are computed in $O(\log m)$ time and $O(m \log m)$ work on the P-CRCW PRAM. □

Given $Lmax_X$ and $Lmin_X$, Algorithm 4 computes order-isomorphism between X and another string.[2]

Lemma 14 *For strings X and Y of equal length m such that $X[1:r] \approx Y[1:r]$ for some $r \le m$, Algorithm 4 computes a (prefix-)tight mismatch position pair in $O(1)$ time and $O(m - r)$ work on the P-CRCW PRAM, given that $Lmax_X$ and $Lmin_X$ are already computed. If $X \approx Y$, it returns zero, i.e., $\langle 0, 0\rangle$.*

[2] The value of the last argument r in the function GetMismatchPos($Lmax_X$, $Lmin_X$, Y, r) is usually 0 except the call from Algorithm 12.

Algorithm 5: Returns a prefix/suffix-tight witness for offset a

```
1 Function PrefixTightWitness(a)
2    ⟨w1, w2⟩ ← GetMismatchPos(Lmax_P, Lmin_P, P[a + 1 : m], 0);
3    return ⟨w1, w2⟩;
4 Function SuffixTightWitness(a)
5    ⟨w1, w2⟩ ← GetMismatchPos(Lmax_rev(P), Lmin_rev(P), rev(P)[a + 1 : m], 0);
6    if ⟨w1, w2⟩ ≠ ⟨0, 0⟩ then ⟨w1, w2⟩ ← ⟨m − w2 − a + 1, m − w1 − a + 1⟩;
7    return ⟨w1, w2⟩;
```

Proof In Algorithm 4, for each position i of X, we "attach" a processor to compute $F_X(Y, i)$ defined in Eq. 1. 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 $i > r$, 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. If $F_X(Y, i) = 0$ for every $i > r$, it means $X \approx Y$, in which case the initial value $\langle 0, 0 \rangle$ of $\langle w_1, w_2 \rangle$ is returned. □

Let us call a witness pair $\langle i, j \rangle \in \mathcal{W}_P(a)$ *prefix/suffix-tight* if it is a prefix/suffix-tight mismatch position pair for $P[: m - a]$ and $P[1 + a :]$. In other words, it is prefix-tight if and only if $j = LIP_P(a) + 1$, and it is suffix-tight if and only if $i = m - LIS_P(a)$.

Using Algorithm 4, one can compute a prefix-tight witness $\langle w_1, w_2 \rangle$ for an arbitrary offset a in $O(1)$ time and $O(m)$ work if $\mathcal{W}_P(a) \neq \emptyset$. The value of $LIP_P(a)$ is then obtained as $w_2 - 1$. This is valid in the case of $\mathcal{W}_P(a) = \emptyset$ as well.

Symmetrically, one can compute suffix-tight witnesses based on the fact that $\langle i, j \rangle$ is a prefix-tight mismatching position pair for X and Y iff $\langle |X| - j + 1, |X| - i + 1 \rangle$ is a suffix-tight mismatching position pair for $\mathrm{rev}(X)$ and $\mathrm{rev}(Y)$. The procedures for computing prefix/suffix-tight witnesses are described in Algorithm 5. Note that $\mathrm{rev}(P)$ can easily be computed from P by $O(1)$ time and $O(m)$ work, and thus the cost for computing $Lmax_{\mathrm{rev}(P)}$ and $Lmin_{\mathrm{rev}(P)}$ is the same for computing $Lmax_P$ and $Lmin_P$.

In the sequel, by a *tight witness*, we refer to a prefix-tight witness.

4.1 Parallel pattern preprocessing

The goal of the preprocessing stage is to compute a witness table $W[0 : m - 1]$ for P. Here, for technical convenience, we prepend zero to the definition of a witness table introduced in Sect. 3 so that $W[0] = \langle 0, 0 \rangle$. Still, we have $W[a] = \langle 0, 0 \rangle$ if $\mathcal{W}_P(a) = \emptyset$, and $W[a] \in \mathcal{W}_P(a)$ otherwise. One can compute a witness table naively calling either of the functions of Algorithm 5 for all the offsets $a < m$. However, this naive method costs as much as $\Omega(m^2)$ work. We will present a more efficient algorithm in this subsection.

Our pattern preprocessing algorithm is described in Algorithm 7 and its outline is illustrated in Fig. 8. Initially, all entries of the witness table are set to zero. At any point of the execution of the preprocessing algorithm, if $W[i]$ is not zero, then it must hold $W[i] \in \mathcal{W}_P(i)$. We say that position i is *finalized* if $W[i] = \langle 0, 0 \rangle$ implies $\mathcal{W}_P(i) = \emptyset$ and $W[i] \neq \langle 0, 0 \rangle$ implies $W[i] \in \mathcal{W}_P(i)$. We call i a *zero position* if $W[i]$ is zero. During the execution of Algorithm 7, the table is divided into two parts. The *head* is a prefix of a certain length and the *tail* is the rest suffix. Let us write the head and the tail at round k of the while-loop by $Head_k$ and $Tail_k$, respectively. Throughout the algorithm execution, the tail part is always finalized. On the other hand, though the zero entries of the head are not necessarily reliable,

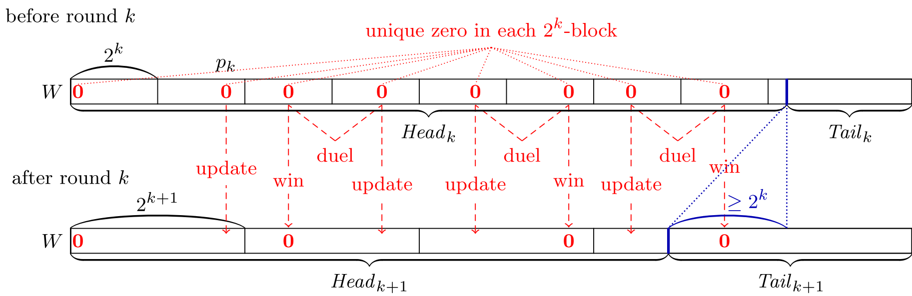


Fig. 8 Illustration of the preprocessing invariant. W is partitioned into head and tail. The head is 2^k-sparse and the tail is finalized. Here, **0** indicates zero entries $\langle 0, 0\rangle$. The 2^k-sparsity is achieved by duels. The tail grows by at least 2^k at each round

such zero positions become fewer and fewer. Consider partitioning the head 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 $i = 0, \dots, \lfloor h_k/2^k \rfloor - 1$ and $W[\lfloor h/2^k \rfloor \cdot 2^k : h_k - 1]$ where $h_k = |Head_k|$ is the size of the head. We say that $W[0 : x]$ is 2^k-*sparse* if every 2^k-block of $W[0 : x]$ contains exactly one zero position possibly except that the last 2^k-block has none. We will guarantee that $Head_k$ is 2^k-sparse. Note that when the head is 2^k-sparse, the unique zero position of the first 2^k-block $W[0 : 2^k - 1]$ is always 0 ($W[0] = \langle 0, 0\rangle$) and $W[1 : 2^k - 1]$ contains no zeros.

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 $W[p_k] = \langle 0, 0\rangle$. Then, we let $Head_{k+1} = W[0 : m-t-1]$ and $Tail_{k+1} = W[m-t : m-1]$ for $t = |Tail_{k+1}| = \max(|Tail_k| + 2^k,\ LIP_P(p_k))$. That is, the tail is expanded at least by 2^k. When $|Head_k| < 2^k$, the 2^k-sparsity means that all the positions in the witness table are finalized. So, Algorithm 7 exits the while loop and halts.

The goal of this subsection is to show Algorithm 7 computes a witness table in $O(\log m)$ time and $O(m \log m)$ work on the P-CRCW PRAM (Theorem 24). In the remainder of this subsection, we explain how to maintain the 2^k-sparsity of the head and finalize the tail. Before going into the detail, we prepare a technical function `GetZeros(l, r, k)` in Algorithm 6, which returns the zero positions $i \in \{l, \dots, r\}$ in the witness table, assuming that $W[0 : r]$ satisfies the 2^k-sparsity. Due to the 2^k-sparsity, each 2^k-block has just one zero position. Thus, it returns an array of length $\lfloor r/2^k \rfloor - \lfloor l/2^k \rfloor + 1$ each of whose entries has the unique zero position of the corresponding 2^k-block. The first and the last 2^k-blocks in $W[l : r]$ may be incomplete and their zero positions would be outside $W[l : r]$, in which case the corresponding entry will be -1. Algorithm 6 runs in $O(1)$ time and $O(r - l)$ work on the P-CRCW PRAM. We note that at Line 4, the assignment operation is denoted by "$\Leftarrow$", since the array A is global and accessible for every processor, but there will be just one processor that accesses $A[j]$ for each j under the assumption of the 2^k-sparsity.

4.1.1 Head maintenance

First we discuss how the algorithm makes $Head_k$ 2^k-sparse. We maintain the head so that at the beginning of round k of Algorithm 7, it satisfies the following invariant properties.

- $Head_k$ is 2^k-sparse.

Algorithm 6: 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, 0⟩ then A[⌊i/2^k⌋ − ⌊l/2^k⌋] ⇐ i ;
5     return A;
```

Algorithm 7: Parallel algorithm for the pattern preprocessing

```
1 Function PreprocessingParallel()
2     compute Lmax_P and Lmin_P;          /* Those are used in PrefixTightWitness */
3     compute Lmax_rev(P) and Lmin_rev(P); /* Those are used in SuffixTightWitness
        */
4     head ← m, k ← 0;                     /* head is the size of Head_k */
5     while 2^k ≤ head do
6         p ← GetZeros(2^k, 2^{k+1} − 1, k)[0];
7         W[p] ← PrefixTightWitness(p);
8         if W[p] = ⟨0, 0⟩ then
9             lcp ← m − p
10        else
11            lcp ← max W[p] − 1;
12        old_head ← head;
13        head ← min(old_head − 2^k, m − lcp);
14        SatisfyHeadSparsity(head − 1, k);
15        FinalizeTail(head, old_head, p, k);
16        k ← k + 1;
```

Algorithm 8: Satisfy 2^{k+1}-sparsity of $Head_{k+1}$, whose ending position is given as the first argument h

```
1 Function SatisfyHeadSparsity(h, k)
2     A ← GetZeros(2^{k+1}, h, 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             w ← W[j2 − j1];                /* W[j2 − j1] is not zero */
7             if P[w] ≉ P[w ⊕ j2] then
8                 W[j2] ⇐ w;
9             else
10                W[j1] ⇐ w ⊕ (j2 − j1);
```

- For all non-zero positions i of $Head_k$,
  - $W[i] \in \mathcal{W}_P(i)$,
  - $\max W[i] \leq |Tail_k| + 2^k$.

The head maintenance procedure SatisfyHeadSparsity is described in Algorithm 8. Before calling the function SatisfyHeadSparsity, Algorithm 7 finalizes the suspected period p_k, the first position after 0 such that $W[p_k] = \langle 0, 0\rangle$. Due to the 2^k-sparsity,

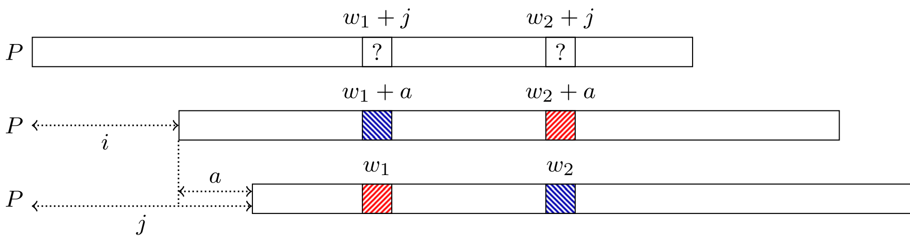


Fig. 9 Dueling with respect to the pattern between offsets i and $j = i + a$. Under the assumption that $w = \langle w_1, w_2 \rangle \in \mathcal{W}_P(a)$, i.e., $P[w] \not\approx P[w \oplus a]$, by comparing $P[w]$ and $P[w \oplus j]$, one can find a witness for either i or j, as long as $j + w_2 \leq m$

$2^k \leq p_k < 2^{k+1}$. Algorithm 7 finds the suspected period p_k at Line 6 and then finalizes the position p_k at Line 7.

Let us explain how Algorithm 8 works. The task of `SatisfyHeadSparsity`(h, k) is to make $W[0 : h]$ satisfy the 2^{k+1}-sparsity. 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 $head \leq m - LIP_P(p_k) = p_k < 2^{k+1}$ when Algorithm 7 calls `SatisfyHeadSparsity(`$head - 1, k$`)`. Then the array A obtained at Line 2 is empty and `SatisfyHeadSparsity(`$head - 1, k$`)` does nothing. After `FinalizeTail(`$head, old_head, p, k$`)` finalizes $Tail_{k+1}$, which will be explained in the next subsubsection, the algorithm will halt without going into the next loop, since $|Head_{k+1}| \leq m - LIP_P(p_k) = p_k < 2^{k+1}$. At that moment all positions of W are finalized.

Hereafter we consider the case where p_k is not a period of P, i.e., $\mathcal{W}_P(p_k) \neq \emptyset$. When `SatisfyHeadSparsity(`$head, k$`)` is called, the value of $W[p_k]$ is a tight witness and the first 2^{k+1}-block contains no zeros except $W[0]$. At that moment, the other part of the head is 2^k-sparse. To make it 2^{k+1}-sparse, we perform *duels* between two zero positions i and j $(i < j)$ within each of the 2^{k+1}-blocks of the head except for the first block. The duel w.r.t. the pattern is same as 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 itself. In a duel between two text positions, the loser candidate is eliminated. In a duel between two pattern positions, the loser offset gets a witness. The following lemma shows when and how a duel w.r.t. the pattern can be performed.

Lemma 15 Figure 9 For two offsets i and j with $i < j$, suppose $w \in \mathcal{W}_P(j - i)$ and $j + \max w \leq m$. Then,

- if the offset i survives the duel, i.e., $P[w] \not\approx P[w \oplus j]$, then $w \in \mathcal{W}_P(j)$;
- if the offset j survives the duel, i.e., $P[w] \approx P[w \oplus j]$, then $w \oplus (j - i) \in \mathcal{W}_P(i)$.

Proof If $P[w] \not\approx P[w \oplus j]$, then $w \in \mathcal{W}_P(j)$ by definition. Suppose $P[w] \approx P[w \oplus j]$ and let $a = j - i$. The fact $w \in \mathcal{W}_P(a)$ means $P[w] \not\approx P[w \oplus a]$ and thus $P[w \oplus a] \not\approx P[w \oplus j] = P[(w \oplus a) \oplus i]$, which means $w \oplus a \in \mathcal{W}_P(i)$. □

In our algorithm, the witness used for the duel between i and j in the same 2^{k+1}-block is $W[a]$ for $a = j - i$, which is in the first 2^{k+1}-block. Lemma 17 below ensures that indeed our dueling pairs satisfy the condition of Lemma 15.

Lemma 16 *Suppose the preprocessing invariants hold true at the beginning of round k and $\mathcal{W}_P(p_k) \neq \emptyset$. Then, after Line 7 is executed,* $\max W[a] \leq |Tail_{k+1}| + 1$ *holds for any position a in the first 2^{k+1}-block.*

Proof Recall that $|Tail_{k+1}| = \max(|Tail_k| + 2^k, LIP_P(p_k))$. If $a \neq p_k$, by the invariant property of the previous round, $W[a] \neq \langle 0, 0\rangle$ and $\max W[a] \leq |Tail_k| + 2^k \leq |Tail_{k+1}|$. If $a = p_k$, $W[a]$ holds a tight witness for offset p_k (Algorithm 7, Line 7), i.e., $\max W[a] = LIP_P(p_k) + 1 \leq |Tail_{k+1}| + 1$. □

Lemma 17 *For any positions i, j of $Head_{k+1}$ such that $0 < j - i < 2^{k+1}$, it holds that $j + \max W[j-i] \leq m$ when* `SatisfyHeadSparsity`$(head, k)$ *is called at Line 14 in Algorithm 7.*

Proof Since j is in $Head_{k+1}$, we have $j \leq |Head_{k+1}| - 1$. By Lemma 16, $j + \max W[j-i] \leq |Head_{k+1}| - 1 + |Tail_{k+1}| + 1 = m$. □

Therefore, every pair of offsets in the same block can perform a duel in the execution of `SatisfyHeadSparsity`$(head, k)$ using the first 2^{k+1}-block of the witness table and the loser will get a witness by Lemma 15. It remains to show the invariant property is certainly maintained. We note that `FinalizeTail`$(head, old_head, p, k)$ does not modify the head at all.

Lemma 18 *At the beginning of round k, the invariant property holds for $Head_k$.*

Proof We show the lemma by induction on k. For $k = 0$, every position is zero, so the lemma vacuously holds. We show the lemma holds for $k + 1$ assuming that it is the case for k. The 2^k-sparsity and the witness value correctness are followed from Lemmas 15 to 17. It remains to show $\max W[i] \leq |Tail_{k+1}| + 2^{k+1}$ for all non-zero positions of $Head_{k+1}$. Concerning positions a in the first 2^{k+1}-block, Lemma 16 shows a stronger property: $W[a] \leq |Tail_{k+1}| + 1$. So, it suffices to show the claim for positions belonging to other blocks. If $W[i]$ is not updated from the previous round, then $\max W[i] \leq |Tail_k| + 2^k \leq |Tail_{k+1}|$ by the induction hypothesis. Suppose $W[i]$ was zero in the previous round and has been updated by losing the duel against another offset j in the same 2^{k+1}-block. For $a = |i - j|$, the algorithm lets $W[i] = W[a]$ or $W[i] = W[a] \oplus a$. In either case, $\max W[i] \leq \max W[a] + a \leq |Tail_{k+1}| + 1 + a \leq |Tail_{k+1}| + 2^{k+1}$. □

Lemma 19 *Algorithm 8 runs in $O(1)$ time and $O(m/2^k)$ work on P-CRCW-PRAM.*

Proof `GetZeros`$(2^{k+1}, h, k)$ requires $O(1)$ time and $O(m/2^k)$ work. We then use $\lfloor (h - 2^{k+1})/2^{k+1} \rfloor \in O(m/2^k)$ processors in parallel, each of which runs in constant time. □

4.1.2 Tail finalization

Next, we discuss how we finalize $Tail_{k+1}$ for the round k using Algorithm 9. Since $Tail_k$ is finalized at the beginning of round k, we only need to finalize positions of $Tail_{k+1}$ which are not in $Tail_k$. Let $\mathcal{T} = \{\, |Head_{k+1}|, \ldots, |Head_k| - 1 \,\}$ be the positions to finalize. We call $\mathcal{T}$ *small* if $|\mathcal{T}| \leq p_k$. Since $p_k < 2^{k+1}$, when $\mathcal{T}$ is small, due to the 2^k-sparsity, there are at most three zero positions to finalize. In this case, we can naively call `PrefixTightWitness`(i) to finalize $W[i]$ for those zero positions i by finding them using `GetZeros`. This case is handled in Lines 2 to 6.

On the other hand, when $\mathcal{T}$ is not small, we need a more elaborated technique for efficient finalization. Note that, $|\mathcal{T}| = |Tail_{k+1}| - |Tail_k| > p_k$ implies $|Tail_{k+1}| = \max(|Tail_k| + 2^k, LIP_P(p_k)) = LIP_P(p_k)$. In this case, we partition $\mathcal{T}$ into non-empty subsets $\mathcal{T}_0, \ldots, \mathcal{T}_{p_k - 1}$ so that $\mathcal{T}_s$ consists of positions $i \equiv s \pmod{p_k}$. The tail finalization

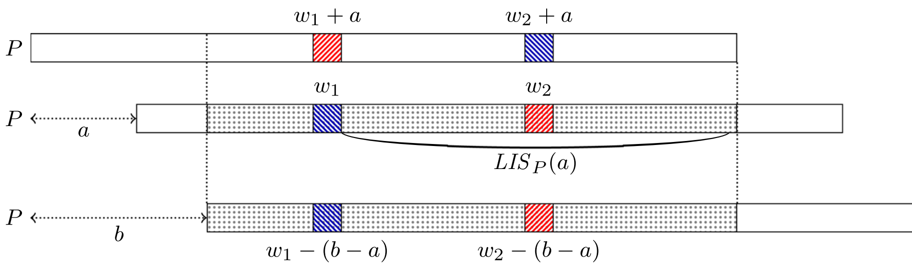


Fig. 10 Illustration to Lemma 22 when $m - b > LIS_P(a)$. The dotted regions are isomorphic $P[1 : m - b] \approx P[b - a + 1 : m - a]$ by Lemma 20. The shaded positions show the mismatch between $P[w_1, w_2]$ and $P[w_1 + a, w_2 + a]$

is performed on each $\mathcal{T}_s$ independently. We will pick a referential position x_s for each $\mathcal{T}_s$ and update every position i in $\mathcal{T}_s$ using a witness of the referential position x_s with the appropriate shift ($\oplus(x_s - i)$, which may be negative). Let $q_s = \max \mathcal{T}_s$ and $r_s = \min \mathcal{T}_s$. Thanks to the 2^k-sparsity, $W[q_s]$ is not zero for most bags $\mathcal{T}_s$. We use this non-zero witness $W[q_s]$ as the reference for finalizing all the other positions in $\mathcal{T}_s$, based on Lemma 21 shown below. There are at most three exceptional bags $\mathcal{T}_s$ where $W[q_s]$ is zero due to $p_k < 2^{k+1}$. For those bags, we compute a suffix-tight witness for r_s and use it as the reference, based on Lemma 22.

Lemma 20 *Given positions $a, b \in \mathcal{T}_s$ such that $a < b$, $P[1 : m - b] \approx P[b - a + 1 : m - a]$.*

Proof Let $\ell = LIP_P(p_k) = |Tail_{k+1}|$.

Since a and b are positions inside the tail, we have $m - \ell \leq a < b < m$. Since p_k is a period of $P[1 : \ell]$ and $(b - a)$ is a multiple of p_k, by Lemma 5, $(b - a)$ is also a period of $P[1 : \ell]$, i.e., $P[1 : \ell - (b - a)] \approx P[1 + (b - a) : \ell]$. Taking the prefixes of length $m - b \leq \ell - (b - a)$ of those isomorphic strings, we obtain $P[1 : m - b] \approx P[1 + b - a : m - a]$. □

Lemma 21 *Suppose $\mathcal{W}_P(q_s) \neq \emptyset$ for $q_s = \max \mathcal{T}_s$. For any position $i \in \mathcal{T}_s$ and any witness $w \in \mathcal{W}_P(q_s)$, we have $w \oplus (q_s - i) \in \mathcal{W}_P(i)$.*

Proof For $w \in \mathcal{W}_P(q_s)$, $P[w \oplus q_s] \not\approx P[w]$. By Lemma 20, $P[q_s - i + 1 : m - i] \approx P[1 : m - q_s]$, which implies $P[w \oplus (q_s - i)] \approx P[w]$. Therefore, $P[(w \oplus (q_s - i)) \oplus i] \not\approx P[w \oplus (q_s - i)]$. □

Based on Lemma 21, the algorithm finalizes $W[i]$ for $i \in \mathcal{T}_s$ with $W[q_s] \neq \langle 0, 0 \rangle$ in the **for each** parallel computation of Line 9.

Now, let us consider the case where $W[q_s] = \langle 0, 0 \rangle$. This case is more involved than the previous case. The algorithm uses the following lemma to finalize $W[i]$ for $i \in \mathcal{T}_s$ with $W[q_s] = \langle 0, 0 \rangle$ efficiently.

Lemma 22 *Given offsets $a, b \in \mathcal{T}_s$ such that $a < b$, $\mathcal{W}_P(b) = \emptyset$ iff $m - b \leq LIS_P(a)$* Figure 10. If $\mathcal{W}_P(b) \neq \emptyset$, then $\mathcal{W}_P(a) \neq \emptyset$ and $w \ominus (b - a) \in \mathcal{W}_P(b)$ for any suffix-tight witness w for offset a.

Proof Lemma 20 implies $P[1 : m - b] \approx P[b - a + 1 : m - a]$. The first half of the lemma follows from

$$\begin{aligned} \mathcal{W}_P(b) = \emptyset &\iff P[1 : m - b] \approx P[b + 1 : m] \\ &\iff P[b - a + 1 : m - a] \approx P[b + 1 : m] \iff LIS_P(a) \geq m - b\,, \end{aligned}$$

Algorithm 9: Finalize $Tail_{k+1}$

```
1  Function FinalizeTail(head, old_head, p, k)
2    if old_head − head ≤ p then
3      A ← GetZeros(head, old_head − 1, k);                 /* |A| ≤ 3 */
4      for z = 0 to |A| − 1 do
5        i ← A[z];
6        if i ≠ −1 then W[i] ← PrefixTightWitness(i);
7    else
8      T ← {head, ..., old_head − 1};
9      for each i ∈ T do in parallel
10       q ← max{x ∈ T | x ≡ i (mod p)};
11       if W[q] ≠ ⟨0, 0⟩ then W[i] ⇐ W[q] ⊕ (q − i);
12     A ← GetZeros(old_head − p, old_head − 1, k);         /* |A| ≤ 3 */
13     for z = 0 to |A| − 1 do
14       q ← A[z];
15       if q ≠ −1 then
16         r ← min{x ∈ T | x ≡ q (mod p)};
17         W[r] ← SuffixTightWitness(r);
18         lcs ← m − r − min W[r];
19         for each i ∈ T do in parallel
20           if i ≡ r (mod p) and m − i > lcs then
21             W[i] ← W[r] ⊖ (i − r);
```

where the last equivalence holds by the definition of $LIS_P(a)$.

Now, we prove the second half of the lemma. When $\mathcal{W}_P(b) \neq \emptyset$, by the first half of the lemma, we have $LIS_P(a) < m - b < m - a$. Thus, the offset a has a suffix-tight witness $w = \langle w_1, w_2 \rangle$ such that $w_1 = m - a - LIS_P(a) > b - a$. By definition, $P[w] \not\approx P[w \oplus a]$. On the other hand, $P[1 : m - b] \approx P[b - a + 1 : m - a]$ implies $P[w] \approx P[w \ominus (b - a)]$, where $w \ominus (b - a)$ is a pair of positive integers by $w_1 > b - a$.

Hence, $P[w \ominus (b-a)] \not\approx P[w \oplus a] = P[(w \ominus (b-a)) \oplus b]$, i.e., $w \ominus (b-a) \in \mathcal{W}_P(b)$. □

For bags $\mathcal{T}_s$ such that $W[q_s] = \langle 0, 0 \rangle$, Algorithm 9 finalizes positions $i \in \mathcal{T}_s$ at Lines 12–21 based on Lemma 22. It first computes $LIS_P(r_s)$ for $r_s = \min \mathcal{T}_s$ and a suffix-tight witness w for offset r_s unless $\mathcal{W}_P(r_s) = \emptyset$. Then, for $i \in \mathcal{T}_s$ such that $m - i > LIS_P(r_s)$, the algorithm updates $W[i]$ to $w \ominus (i - r_s)$. Note that if $\mathcal{W}_P(r_s)$ is empty, so is $\mathcal{W}_P(i)$ by Lemma 22.

Lemma 23 *For the round k, Algorithm 9 finalizes $Tail_{k+1}$ in $O(1)$ time and $O(m)$ work on P-CRCW PRAM.*

Proof Let t be the size difference of $Tail_k$ and $Tail_{k+1}$. First we consider the case when $t \leq p_k$. Since $Head_k$ satisfies the 2^k-sparsity, there are at most three zero positions in $W[|Head_{k+1}| : |Head_k|]$. Each such position can be finalized in $O(1)$ time and $O(m)$ work.

Let us consider the case when $t > p_k$.

Obviously, the parallel computation of Line 9 costs $O(1)$ time and $O(t)$ work.

The function call GetZeros($old_head - p, old_head - 1, k$) at Line 12 costs $O(1)$ time and $O(t)$ work. The for-loop at Line 13 is repeated at most three times, since $Head_k$ is 2^k-sparse and $2^k \leq p_k < 2^{k+1}$. The algorithm computes $LIS_P(r_s)$ at Line 17 in $O(1)$ time and $O(m)$ work. Then, the parallel computation at Line 19 costs $O(1)$ time and $O(t)$ work. Thus, overall Algorithm 9 runs in $O(1)$ time and $O(m)$ work. □

4.1.3 Summary of pattern preprocessing

Theorem 24 *Algorithm 7 computes a witness table in $O(\log m)$ time and $O(m \log m)$ work on the P-CRCW PRAM.*

Proof By Lemmas 18, 19, and 23, together with the fact that Algorithm 7 repeats the while-loop at most $\lceil \log m \rceil$ times. □

4.2 Parallel pattern searching

Our pattern searching algorithm prunes candidates of the text T of length n in two stages: dueling and sweeping stages. During the dueling stage, candidates duel with each other, until the surviving candidates 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 use a Boolean array $C[1 : n - m + 1]$ and initialize every entry of C to *True* at the beginning. If a candidate T_i gets eliminated, we set $C[i] = \mathit{False}$. The pattern searching algorithm updates C in such a way that $C[i] = \mathit{True}$ iff i is a pattern occurrence. Entries of C are updated at most once during the dueling and sweeping stages. Hereinafter, we denote the number of candidates by $n' = n - m + 1$.

4.2.1 Dueling stage

The dueling stage is described in Algorithm 11. Recall that x is consistent with $x + a$ if $\mathcal{W}_P(a) = \emptyset$ or $a \geq m$. We say that a set of positions is *consistent* if all elements in the set are pairwise consistent. During the round k, the algorithm partitions the candidate positions 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 11 is as follows.

- At any point of execution of Algorithm 11, all pattern occurrences survive.
- For round k, each $\mathcal{C}_{k,j}$ is consistent.

The survivor set $\mathcal{C}_{k,j}$ is obtained by "merging" $\mathcal{C}_{k-1,2j-1}$ and $\mathcal{C}_{k-1,2j}$, where $\mathcal{C}_{k,j}$ 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 $\mathcal{C}_{k-1,2j-1} \cup \mathcal{C}_{k-1,2j}$. At the end of the dueling stage, $\mathcal{C}_{\lceil \log n' \rceil,1}$ is a consistent set including all the occurrence positions. We then let $C[i] = \mathit{True}$ iff $i \in \mathcal{C}_{\lceil \log n' \rceil,1}$. In our algorithm, each set $\mathcal{C}_{k,j}$ is represented as an integer array, where elements are sorted in increasing order. We will represent the i-th smallest element of an integer set $\mathcal{C}$ by $\mathcal{C}[i]$.

Let us consider merging two respectively consistent sets $\mathcal{A}(= \mathcal{C}_{k-1,2j-1})$ and $\mathcal{B}(= \mathcal{C}_{k-1,2j})$, where $\mathcal{A}$ *precedes* $\mathcal{B}$, i.e., $\max \mathcal{A} < \min \mathcal{B}$. We must find a consistent set $\mathcal{C}$ such that $\widehat{\mathcal{A}} \cup \widehat{\mathcal{B}} \subseteq \mathcal{C} \subseteq \mathcal{A} \cup \mathcal{B}$ where $\widehat{\mathcal{A}} = \{ a \in \mathcal{A} \mid T_a \approx P \}$ and $\widehat{\mathcal{B}} = \{ b \in \mathcal{B} \mid T_b \approx P \}$ are the sets of occurrences of P in $\mathcal{A}$ and $\mathcal{B}$, respectively.

Lemma 25 *Suppose that we are given two respectively consistent position sets $\mathcal{A}$ and $\mathcal{B}$ such that $\mathcal{A}$ precedes $\mathcal{B}$. If $a \in \mathcal{A}$ and $b \in \mathcal{B}$ are consistent, then $\mathcal{A}_{\leq a} \cup \mathcal{B}_{\geq b}$ is also consistent, where $\mathcal{A}_{\leq a} = \{ i \in \mathcal{A} \mid i \leq a \}$ and $\mathcal{B}_{\geq b} = \{ j \in \mathcal{B} \mid j \geq b \}$.*

Proof Let $i \in \mathcal{A}_{\leq a}$ and $j \in \mathcal{B}_{\geq b}$. Since candidate pairs of i and a, a and b, b and j are respectively consistent, i is consistent with j by Lemma 6. □

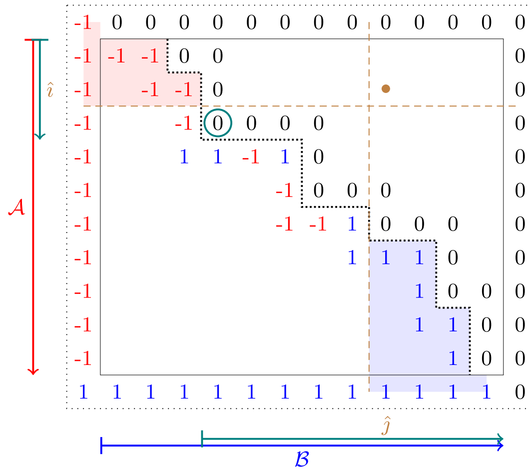


Fig. 11 Padded grid G given two consistent sets $\mathcal{A}$ and $\mathcal{B}$. The grid is separated into the zero region and the non-zero region by the dotted boundary line (Lemma 25). The coordinate $(\hat{\imath}, \hat{\jmath})$ is indicated by the brown dot. The red- and blue-shaded areas consist of -1 and 1 only, respectively (Lemma 26). Our algorithm outputs (i, j) such that $G[i][j] = 0$, $G[i][j-1] = -1$, $G[i+1][j'] = 1$, and $G[i+1][j'+1] = 0$ for some j'. If there are more than one such coordinate, the smallest i will be chosen by the priority. The output coordinate is indicated by the green circle above. Then the obtained set consists of the elements represented by the two green arrows

Therefore, it suffices to find $(a, b) \in (\mathcal{A} \cup \{-\infty\}) \times (\mathcal{B} \cup \{\infty\})$ such that $a \geq \max(\widehat{\mathcal{A}} \cup \{-\infty\})$, $b \leq \min(\widehat{\mathcal{B}} \cup \{\infty\})$, and a and b are consistent, where we assume ∞ and $-\infty$ are consistent with any other positions. Then, $\mathcal{A}_{\leq a} \cup \mathcal{B}_{\geq b}$ has the desired property. Indeed, $\hat{a} = \max(\widehat{\mathcal{A}} \cup \{-\infty\})$ and $\hat{b} = \min(\widehat{\mathcal{B}} \cup \{\infty\})$ satisfy the property, but our goal at this stage is a little more relaxed.

To find such a pair (a, b), let us consider a grid G of size $(|\mathcal{A}|+2) \times (|\mathcal{B}|+2)$. Figure 11 illustrates the grid, where indices of $\mathcal{A}$ and $\mathcal{B}$ are presented along the directions of rows and columns, respectively. For $1 \leq i \leq |\mathcal{A}|$ and $1 \leq j \leq |\mathcal{B}|$, $G[i][j]$ represents the result of the duel between $\mathcal{A}[i]$ and $\mathcal{B}[j]$ using the witness table W, which are the i-th and j-th smallest elements of $\mathcal{A}$ and $\mathcal{B}$, respectively. We define $G[i][j] = 0$ if $W[d] = 0$ for $d = \mathcal{B}[j] - \mathcal{A}[i]$. If $W[d] \neq 0$ and $\mathcal{A}[i]$ wins the duel, then $G[i][j] = -1$. Otherwise, $\mathcal{B}[j]$ wins the duel and $G[i][j] = 1$. For the sake of explanatory 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, $G[i][0] = -1$ for $i \in \{0, \ldots, |\mathcal{A}|\}$, $G[|\mathcal{A}|+1][j] = 1$ for $j \in \{0, \ldots, |\mathcal{B}|\}$, $G[i][|\mathcal{B}|+1] = 0$ for $i \in \{1, \ldots, |\mathcal{A}|+1\}$, and $G[0][j] = 0$ for $j \in \{1, \ldots, |\mathcal{B}|+1\}$. We will not compute the whole G, but this concept helps to understand the behavior of our algorithm.

In terms of the grid representation, our goal is to find a coordinate (i, j) such that $G[i][j] = 0$ and it is to the lower left of $(\hat{\imath}, \hat{\jmath})$ (brown dot in Fig. 11) where $\hat{\imath} = \max(\{i' \mid \mathcal{A}[i'] \in \widehat{\mathcal{A}}\} \cup \{0\})$ and $\hat{\jmath} = \min(\{j' \mid \mathcal{B}[j'] \in \widehat{\mathcal{B}}\} \cup \{|\mathcal{B}|+1\})$. Then, $\mathcal{A}_{\leq \mathcal{A}[i]} \cup \mathcal{B}_{\geq \mathcal{B}[j]}$ has the desired property, where we assume $\mathcal{A}[0] = -\infty$ and $\mathcal{B}[|\mathcal{B}|+1] = \infty$.

Lemma 25 implies that if $G[i][j] = 0$ then $G[i'][j'] = 0$ for any $i' \leq i$ and $j' \geq j$. Therefore, grid G can be divided into two regions: the upper-right region that consists of only 0 and the rest that consists of a mixture of -1 and 1. The boundary line looks like a step function. The distributions of 1 and -1 in the non-zero region are not totally random. Since

Algorithm 10: Merge two consistent sets $\mathcal{A}$ and $\mathcal{B}$

```
1  Function Merge(A, B)
2    for each i ∈ {0, ..., |A| + 1} do in parallel
3      j' ← 0; j'' ← |B| + 1;
4      while j'' − j' > 1 do
5        j ← ⌊(j' + j'')/2⌋;
6        if G[i][j] = 0 then j'' ← j;
7        else j' ← j;
8      D[i] ⇐ j';
9    for each i ∈ {0, ..., |A|} do in parallel
10     if G[i][D[i]] = −1 and G[i + 1][D[i + 1]] = 1 then
11       i_* ⇐ i; j_* ⇐ D[i] + 1;
12   return (i_*, j_*);
```

occurrences will never lose the duel, if $\mathcal{A}[i] \in \widehat{\mathcal{A}}$, then row i consists of non-positive elements only, and if $\mathcal{B}[j] \in \widehat{\mathcal{B}}$, then column j consists of non-negative elements only. Particularly, $G[\hat{\imath}][\hat{\jmath}] = 0$. The following lemma strengthens this observation.

Lemma 26 *If $\widehat{\mathcal{A}} \neq \emptyset$ and $\mathcal{A}[i] \leq \max \widehat{\mathcal{A}}$, then row i consists only of non-positive elements. Similarly, if $\widehat{\mathcal{B}} \neq \emptyset$ and $\mathcal{B}[j] \leq \min \widehat{\mathcal{B}}$, then column j consists only of non-negative elements.*

Proof We prove the first half of the lemma. The second claim can be proven in the same way. We show that if $i \leq \hat{\imath}$ and $G[i][j] \neq 0$, then $G[i][j] = -1$ for any $1 \leq j \leq |\mathcal{B}|$. Let $a = \mathcal{A}[i]$, $\hat{a} = \max \widehat{\mathcal{A}} = \mathcal{A}[\hat{\imath}]$, and $b = \mathcal{B}[j]$, and suppose the inconsistency between a and b is witnessed by $W[b-a] = \langle w_1, w_2 \rangle \neq \langle 0, 0 \rangle$, i.e., $P[w_1, w_2] \not\approx P_{b-a+1}[w_1, w_2]$. Since $T_{\hat{a}} \approx P$, $T[b : m + \hat{a} - 1] \approx P[b - \hat{a} + 1 : m]$, which implies $T_b[w_1, w_2] \approx P_{b-\hat{a}+1}[w_1, w_2]$. On the other hand, since a and $\hat{a}$ are consistent, i.e., $P[1 : m - (\hat{a} - a)] \approx P[\hat{a} - a + 1 : m]$, we have $P[b - \hat{a} + 1 : m - (\hat{a} - a)] \approx P[b - a + 1 : m]$, which implies $P_{b-\hat{a}+1}[w_1, w_2] \approx P_{b-a+1}[w_1, w_2]$. Therefore, $T_b[w_1, w_2] \approx P_{b-\hat{a}+1}[w_1, w_2] \approx P_{b-a+1}[w_1, w_2] \not\approx P[w_1, w_2]$. Hence, a wins the duel against b and thus $G[i][j] = -1$. □

Algorithm 10 firstly finds the unique column j_i for each row i such that $G[i][j_i] \neq 0$ and $G[i][j_i + 1] = 0$. Among those boundary coordinates, the algorithm finds a neighbour pair (i, j_i) and $(i + 1, j_{i+1})$ such that $G[i][j_i] = -1$ and $G[i + 1][j_{i+1}] = 1$. Then, it outputs $(i, j_i + 1)$. Notice that we do not precompute all the values $G[i][j]$ of the grid. Each time the algorithm needs to know the value, it lets the candidates $\mathcal{A}[i]$ and $\mathcal{B}[j]$ duel (unless $i \in \{0, |\mathcal{A}| + 1\}$ or $j \in \{0, |\mathcal{B}| + 1\}$), which can be performed in constant time.

Lemma 27 *Algorithm 10 finds a coordinate (i_*, j_*) such that $i_* \geq \hat{\imath}$, $j_* \leq \hat{\jmath}$, and $G[i_*][j_*] = 0$ in $O(\log |\mathcal{B}|)$ time with $O(|\mathcal{A}| \log |\mathcal{B}|)$ work.*

Proof For each i, the first **for each** parallel computation finds j_i such that $G[i][j_i] \neq 0$ and $G[i][j_i + 1] = 0$ by binary search and lets $D[i] = j_i$. Then, the algorithm finds i such that $G[i][j_i] = -1$ and $G[i + 1][j_{i+1}] = 1$. Since $G[i][j_i] = -1$, by Lemma 26, $j_i < \hat{\jmath}$. Similarly, $G[i + 1][j_{i+1}] = 1$ implies $i + 1 > \hat{\imath}$. Thus, $i_* = i \geq \hat{\imath}$ and $j_* = j_i + 1 \leq \hat{\jmath}$ satisfy the desired property by Lemma 25.

Since a duel takes $O(1)$ time and $O(1)$ work, we obtain the claimed complexity. □

Lemma 28 *Given a witness table for P, Algorithm 11 performs the dueling stage in $O(\log^2 n)$ time and $O(n \log^2 n)$ work on P-CRCW PRAM.*

Algorithm 11: Parallel algorithm for the dueling stage

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

Proof Since the while-loop runs $O(\log n)$ times and each loop takes $O(\log n)$ time by Lemma 27, the overall time complexity is $O(\log^2 n)$. Now, let us look at the work complexity. Concerning each round k of the while-loop of Algorithm 11, `Merge`$(\mathcal{A}, \mathcal{B})$ takes $O(2^k \log n)$ work by Lemma 27 and thus it takes $O((n/2^k) \cdot 2^k \log n) = O(n \log n)$ work. Since k ranges from 0 to $\lceil \log n' \rceil$, the overall work complexity is $O(n \log^2 n)$. □

4.2.2 Sweeping stage

The sweeping stage is described in Algorithm 12. The sweeping stage updates C until $C[i] = \mathit{True}$ iff i is a pattern occurrence. All entries in C are updated at most once. In addition to C, we will create a new integer array $R[1 : n']$. Throughout the sweeping stage, we have the following invariant properties:

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

Recall that in the sweeping stage of our serial algorithm presented in Sect. 3.3, we use $\mathit{LIP}(T_x, P)$ to avoid looking into the same position of the text repeatedly. Once we have obtained the value $\ell = \mathit{LIP}(T_x, P)$, if the next candidate T_{x+a} is not too far in the sense that $a \leq \ell$, we can start the comparison between T_{x+a} and P from the position $\ell - a + 1$ (Figure 7). The sweeping stage of our parallel algorithm uses a similar trick, but we do not compute $\mathit{LIP}(T_x, P)$ for candidates from left to right sequentially. Instead, we compute lower bounds of the values for many candidates in the array R in parallel. A lower bound is useful enough to save computation according to the same argument as above.

For each stage k, the array C (and thereby R) is divided into 2^k-blocks, where each position i belongs to the $\lceil i/2^k \rceil$-th block. Unlike the preprocessing and dueling algorithms, k starts from $\lceil \log n' \rceil$ and decreases at each round until $k = 0$. In each 2^k-block, we pick a position

Algorithm 12: Parallel algorithm for the sweeping stage

```
1  Function SweepingStageParallel(P, T, C)
2      create arrays of integers R[1 : m];
3      initialize elements of R to 0;
4      k ← ⌈log n′⌉;
5      while k ≥ 0 do
6          create array Piv[1 : ⌈n′/2^k⌉];
7          initialize elements of Piv to −1;
8          for each i ∈ {1, ..., n′} do in parallel
               // get the position i with the smallest index in the second
                  half of the ⌈i/2^k⌉-th 2^k-block such that C[i] = True
9              if C[i] = True and ⌈i/2^(k−1)⌉ mod 2 = 0 then
10             Piv[⌈i/2^k⌉] ⇐ i ;
11         for each b ∈ {1, ..., ⌈n′/2^k⌉} do in parallel
12             x ← Piv[b];
13             if x ≠ −1 then
14                 w ← max GetMismatchPos(Lmax_P, Lmin_P, T_x, R[x]);
15                 if w = 0 then R[x] ⇐ m;
16                 else R[x] ⇐ w − 1;
17         for each i ∈ {1, ..., n′} do in parallel
18             x ← Piv[⌈i/2^k⌉];
19             if x ≠ −1 then
20                 a ← i − x;
21                 if i ≤ x and R[x] < a + m then C[i] ← False;
22                 if i > x and C[i] = True then R[i] ← max(R[i], R[x] − a);
23         k ← k − 1;
24     return C;
```

x as a "pivot" and computes $LIP(T_x, P)$. Then, using the value $LIP(T_x, P)$, we update the arrays C and R on other surviving positions in the block.

Let us look at each round in more detail. The pivot $x_{k,b}$ in the b-th 2^k-block of C is the smallest index $x_{k,b}$ in the second half of the 2^k-block such that $C[x_{k,b}] = True$. The pivot $x_{k,b}$ is stored in $Piv[b]$ at the first **for each** parallel computation (Line 8) of Algorithm 12. In case there are no survivors in the second half of the block, the block will not be updated in this round. When $k = 0$, each block has a unique element, which is chosen as a pivot if it is alive.

The second **for each** parallel computation puts $LIP(T_{x_{k,b}}, P)$ into $R[x_{k,b}]$ for each 2^k-block. The invariant $LIP(T_{x_{k,b}}, P) \geq R[x_{k,b}]$ ensures that to obtain $LIP(T_{x_{k,b}}, P)$, it is enough to perform the isomorphism check from the $(R[x_{k,b}] + 1)$-th position using the F-function. That is,

$$\texttt{GetMismatchPos}(Lmax_P, Lmin_P, T_x, R[x]) \qquad \text{(Algorithm 4)}$$

gives a tight mismatch position pair for $T_{x_{k,b}}$ and P when $T_{x_{k,b}} \not\approx P$. Using the obtained tight mismatching pair $\langle w_1, w_2 \rangle$, we let $R[x_{k,b}] = LIP(T_{x_{k,b}}, P) = w_2 - 1$. If $T_{x_{k,b}} \approx P$, we let $R[x_{k,b}] = LIP(T_{x_{k,b}}, P) = m$.

Using the value $R[x_{k,b}]$, we update the arrays on the other surviving positions in the same 2^k-block. Figure 12 describes how the algorithm updates C and R in the third **for each** parallel computation (Line 17). Suppose that $T_{x_{k,b}} \not\approx P$ and $\langle w_1, w_2 \rangle$ is a tight mismatch

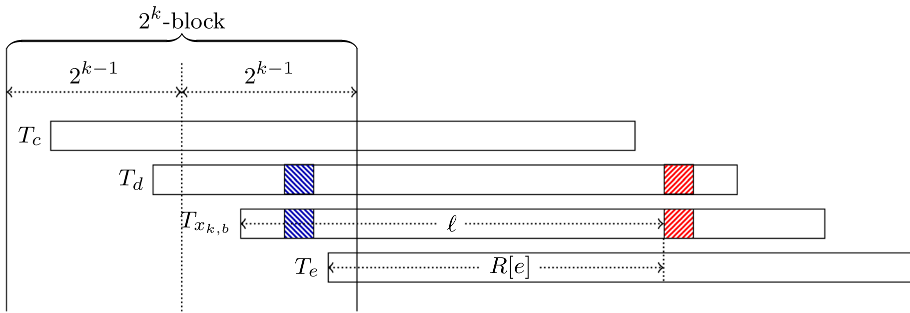


Fig. 12 For each 2^k-block, the algorithm picks a pivot position $x_{k,b}$ and computes $\ell = \mathit{LIP}(T_{x_{k,b}}, P)$. The 2^k-block in this figure contains surviving positions c, d, and e, in addition to $x_{k,b}$, where $c+m \leq x_{k,b}+\ell < d+m$. Concerning the positions left to $x_{k,b}$, since the mismatch position pair is covered by T_d and $T_{x_{k,b}}$, we let $C[d]$ and $C[x_{k,b}]$ be *False*, while $C[c]$ is not updated. Concerning the position e, right to $x_{k,b}$, we set $R[e] = \ell - (e - x_{k,b})$

position pair. Since all surviving candidates are pairwise consistent, any other surviving candidates that "cover" the mismatch position pair cannot match the pattern (Lemma 29). Based on this observation, Algorithm 12 updates $C[i]$ for such candidates T_i in the first half of the 2^k-block at Line 21. On the other hand, at Line 22, the algorithm updates the values of $R[i]$ for indices i in the second half of the block if $C[i] = \mathit{True}$, based on the following observation (Lemma 30). For $\ell = \mathit{LIP}(T_{x_{k,b}}, P)$, the prefixes of $T_{x_{k,b}}$ and P of length ℓ are order-isomorphic. Then, for close neighbor candidates $i > x_{k,b}$, the corresponding prefixes of T_i and P of length $\ell - (i - x_{k,b})$ are also isomorphic, i.e., $\mathit{LIP}(T_i, P) \geq \ell - (i - x_{k,b})$ since $x_{k,b}$ and i are consistent. When i is far from $x_{k,b}$, i.e., $i > x_{k,b} + \ell$, Line 22 does not alter the value $R[i]$. In this way, the algorithm maintains the invariant properties. We note that the algorithm does not update $C[i]$ for i after $x_{k,b}$ or $R[j]$ for j before $x_{k,b}$ within the 2^k-block, but this laziness does not obstruct the maintenance of the invariants.

Lemma 29 *Suppose that $T_j \not\approx P$, for which $w = \langle w_1, w_2 \rangle$ is a mismatch position pair. For any candidate T_i consistent with T_j such that $i < j$, if $w_2 \leq j - i + m$, then $T_i \not\approx P$.*[3]

Proof By $P[w] \not\approx T_j[w] \approx T_i[w \oplus (j - i)]$, we conclude $P \not\approx T_i$. □

Lemma 30 *Suppose that positions j and i are consistent, $T_j[1 : \ell] \approx P[1 : \ell]$, and $\ell > a = i - j > 0$. Then, $\mathit{LIP}(T_i, P) \geq \ell - a$.*

Proof Since j and i are consistent, i.e., $a = i - j$ is a period of P, we have

$$T_i[1 : \ell - a] = T_j[a + 1 : \ell] \approx P[a + 1 : \ell] \approx P[1 : \ell - a] .$$

□

When $k = 0$, all the 2^k-blocks contain just one position x and $R[x]$ is set to be exactly $\mathit{LIP}(T_x, P)$ by Line 22, unless $C[x] = \mathit{False}$ at that time. Then, if $R[x] < m$, then $C[x]$ will be *False* at Line 21. That is, when the algorithm halts, $C[x] = \mathit{True}$ iff $T_x \approx P$.

It remains to show the efficiency of the algorithm. The only nontrivial issue is to estimate the total work amount by the call `GetMismatchPos`$(\mathit{Lmax}_P, \mathit{Lmin}_P, T_x, R[x])$ at Line 14. This call scans the positions of the text from $x + R[x]$ to $x + m - 1$ at maximum for each pivot x. The following lemma implies that those positions are not overlapped.

[3] Actually Lemma 29 holds when $i > j$ and $j - i + 1 \leq w_1$, but we are concerned only with the case where $i < j$.

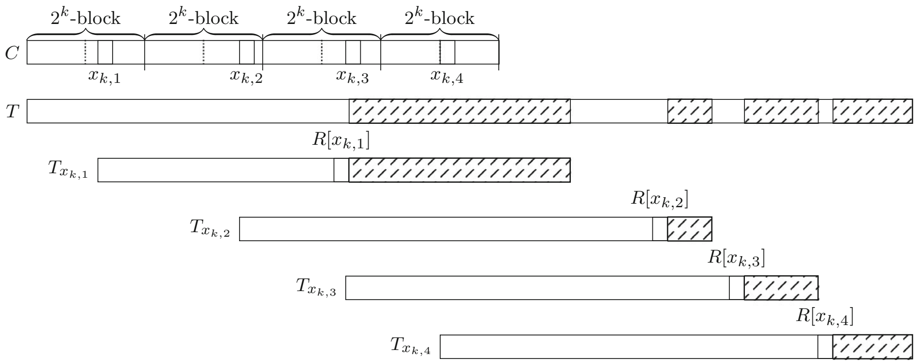


Fig. 13 Relation among pivots of different 2^k-blocks and values of the R-array. The hatched regions of the text are referenced during round k, which do not overlap

Lemma 31 *At the beginning of round k, for two surviving candidate positions i and j with $i < j$ that do not belong to the same 2^k-block, $i + m \leq j + R[j]$.*

Proof At the beginning of round $\lceil \log n' \rceil$, all candidate positions belong to the same $2^{\lceil \log n' \rceil}$-block. Thus, the statement trivially holds (base case). Assuming that the statement holds before round k, we prove that it also holds after round k. Let i and j be surviving positions belonging to different 2^{k-1}-blocks.

First, we consider the case where i and j already belong to different 2^k-blocks. Since elements of the array R are never decremented, the claim holds immediately from the induction hypothesis.

Next, we consider the case where candidate positions i and j belong to the same 2^k-block and then get separated to different 2^{k-1}-blocks. In this case, i and j belong to the first and second halves of the 2^k-block, respectively. Since j is alive, Algorithm 12 successfully chooses a pivot x.

For i to be a surviving candidate after round k, it must be the case that $m+i \leq \mathit{LIP}(T_x, P)+x$ (Line 21). For T_j, Algorithm 12 guarantees $R[j] \geq \mathit{LIP}(T_x, P) - (j - x)$ after round k. Therefore, we obtain

$$m + i \leq \mathit{LIP}(T_x, P) + x \leq R[j] + (j - x) + x = R[j] + j\,.$$

□

Figure 13 shows a particular implication of Lemma 31 when i and j are neighbour pivot positions. The scanned intervals of the parallel calls of `GetMismatchPos(`Lmax_P, Lmin_P, T_i, $R[i]$`)` and `GetMismatchPos(`Lmax_P, Lmin_P, T_j, $R[j]$`)` do not overlap. In other words, during each round, for each i, the **for each** computation of Algorithm 4 is performed at most once. Using the above discussions we have the following regarding the time and work complexities of the sweeping stage.

Lemma 32 *The sweeping stage algorithm runs in $O(\log n)$ time and $O(n \log n)$ work on the P-CRCW PRAM.*

Proof The outer loop of Algorithm 12 runs $O(\log n)$ times. Clearly, the first and the third **for each** parallel computations cost $O(1)$ time and $O(n)$ work in each round. Concerning the second **for each** parallel computations, recall that `GetMismatchPos(`Lmax_P, Lmin_P, T_x, r`)`

costs $O(1)$ time and $O(m-r)$ work (Lemma 14). Thus, for each $b \in \{1, \ldots, \lceil n'/2^k \rceil\}$, if $Piv[b] \neq -1$, the computation costs at most $O(m - R[x_{k,b}]) \subseteq O(x_{k,b} - x_{k,b'})$ work by Lemma 31 where b' is the largest block number such that $b' < b$ and $Piv[b'] \neq -1$. Therefore, the second **for each** parallel computation also costs $O(1)$ time and $O(n)$ work. All in all, the total time is $O(\log n)$ and the total work is $O(n \log n)$. □

4.2.3 Pattern searching theorem

Our pattern searching algorithm runs in $O(\log^2 n)$ time and $O(n \log^2 n)$ work on the P-CRCW PRAM, since the dueling stage (Algorithm 11) takes $O(\log^2 n)$ time and $O(n \log^2 n)$ work by Lemma 28, and the sweeping stage (Algorithm 12) runs in $O(\log n)$ time and $O(n \log n)$ work by Lemma 32. One can improve the time complexity by the standard technique presented at the beginning of this section. That is, we search for pattern occurrences in each substring $T[1:2m-1], T[m+1:3m-1], \ldots, T[km+1:n]$ in parallel, with $k = \lceil \frac{n+1}{m} \rceil - 2$. Then, each of the $k+1$ searches costs $O(\log^2 m)$ time and $O(m \log^2 m)$ work. Therefore, the total amount of work will be $O(n \log^2 m)$.

Theorem 33 *The pattern searching runs in $O(\log^2 m)$ time and $O(n \log^2 m)$ work on the P-CRCW PRAM.*

5 Conclusions and discussion

We have proposed new algorithms for the OPPMP by extending Vishkin's duel-and-sweep algorithm [26] for the exact matching problem. One is serial and the other is parallel. The former runs in linear time, which achieves the theoretical optimum. The latter is the first parallel algorithm for the OPPMP. It runs in $O(\log^2 m)$ time using $O(n \log m)$ work on the P-CRCW PRAM given the text of length n and the pattern of length m. The pattern preprocessing runs in $O(\log m)$ time using $O(m \log m)$ work on the P-CRCW PRAM.

Order-preserving matching is a special case of SCERs and indeed our parallel algorithm is based on the one for the general SCER pattern matching problem by Jargalsaikhan et al. [19, 20]. However, as we discuss in Appendix A in detail, the general algorithm is not suitable for the OPPMP. Our key idea is to use pairs of positions on the input pattern as witnesses for offsets rather than single positions on the encoded input pattern. In addition, the pattern preprocessing of our algorithm takes advantage of the reversibility of OPM, which SCERs do not necessarily satisfy in general, so that it runs faster than the one in [19, 20]. Here, we call a matching relation reversible just in the case where two strings match if and only if so do the reverses of them. While, for example, Cartesian tree matching is not reversible, parameterized matching is reversible. Therefore, the presented technique used in the preprocess can be applied to the pattern matching problems for other reversible SCERs like parameterized matching as well.

Acknowledgements We would like to express our sincere gratitude to the anonymous reviewers for their constructive comments and valuable feedback. Their contributions have been helpful in refining the final version of this paper.This work was supported by JSPS KAKENHI Grant Numbers JP15H05706, JP18K11150, JP19K20208, JP20H05703, and JP21K11745. This work was also supported by ImPACT Program of the Council for Science, Technology and Innovation (Cabinet Office, Government of Japan). Davaajav Jargalsaikhan was supported by a research grant from Tohoku University Division for International Advanced Research and Education.

Author contributions All the co-authors contributed equally to this work.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interests The authors declare no competing interests.

Appendix A. Comparison with the parallel SCER matching algorithms

Our proposed parallel algorithm is largely based on the general matching algorithm for arbitrary SCERs proposed by Jargalsaikhan et al. [19, 20]. An equivalence relation $\cong$ over Σ^* is called an *SCER* just in the case where $X \cong Y$ implies $|X| = |Y|$ and $X[i:j] \cong Y[i:j]$ for any $1 \leq i \leq j \leq |X|$. Clearly, the OPM relation $\approx$ is an SCER. The algorithm in [19] uses an encoding that satisfies the following conditions.

Definition 34 ($\cong$-encoding, [19, Definition 3]) Let Σ and Δ be (possibly infinite) alphabets. We say a function $f : \Sigma^* \to \Delta^*$ is an $\cong$-*encoding* if

(1) $f(X) = f(Y)$ iff $X \cong Y$,
(2) $f(X[1:i]) = f(X)[1:i]$ for any $i \leq |X|$, and
(3) $f(X)[i] = f(Y)[i]$ implies $f(X[j+1:k])[i-j] = f(Y[j+1:k])[i-j]$ for any $j < i \leq k$.

If $X \ncong Y$, one can find a witness position i such that $f(X)[i] \neq f(Y)[i]$. The conditions (2) and (3) imply that when a position witnesses mismatch between substrings of X and Y, then that position witnesses the mismatch between the whole strings X and Y as well. This is an important property to "transfer" witnesses on an offset for other offsets in their algorithm. Accordingly, the efficiency of their algorithm depends on the efficiency of the calculation of the encoding of a string as well as that of recalculating the encoding of a substring from the encoding of the whole string.

Indeed, every SCER $\cong$ admits an encoding satisfying the above: let Δ be the power set of Σ^*, and $f(\varepsilon) = \varepsilon$ and $f(Xc) = f(X) \cdot [Xc]_{\cong}$ for $c \in \Sigma$ where $[Y]_{\cong}$ is the equivalence class of $Y \in \Sigma^*$ under $\cong$ (or, Δ can be any set of symbols that can represent those equivalence classes). This construction guarantees that their algorithm works for every SCER in theory, but computing this encoding is apparently expensive. Actually, many SCERs, like exact match, parameterized match, and Cartesian match, admit computationally cheaper encodings satisfying Definition 34. However, we do not yet know if there is such a reasonable encoding for OPM.

Concerning the OPM relation $\approx$, recall that $X \approx Y$ if and only if $Lmax_X = Lmax_Y$ and $Lmin_X = Lmin_Y$. The encoding $Lmix_X$ of X defined as $Lmix_X[i] = \langle Lmin_X[i], Lmax_X[i] \rangle$ fulfills (1)–(2) of Definition 34, but not (3). For example, consider the mismatch between $X = (4, 6, 1, 5)$ and $Y = (4, 6, 9, 5)$. We have

$$Lmix_X = (\langle 0, 0\rangle,\ \langle 0, 1\rangle,\ \langle 1, 0\rangle,\ \langle 2, 1\rangle)\,,$$
$$Lmix_Y = (\langle 0, 0\rangle,\ \langle 0, 1\rangle,\ \langle 0, 1\rangle,\ \langle 2, 1\rangle)\,.$$

The mismatch is witnessed at position 3, but not at the last position, with this encoding. On the other hand, the mismatch of their suffixes $X' = (1, 5) \not\approx Y' = (9, 5)$ is witnessed at the last position.

$$Lmix_{X'} = (\langle 0, 0\rangle,\ \langle 0, 1\rangle)\,,$$

$$Lmix_{Y'} = (\langle 0, 0\rangle,\ \langle 1, 0\rangle)\,.$$

Therefore, the algorithm proposed in [19] does not work with this encoding *Lmix*. Instead, we have designed an algorithm that uses two positions as a mismatch witness, where we do not compare the encoded characters. This modification exempts us from the encoding costs. Furthermore, the OPM relation $\approx$ is closed under reversal, i.e., $X \approx Y$ iff $\mathrm{rev}(X) \approx \mathrm{rev}(Y)$, which is not guaranteed in general SCERs. Thanks to this property, our proposed pattern preprocessing (Algorithm 9 more specifically) runs faster than the one for general SCERs in [19].

References

1. Amir, A., Kondratovsky, E.: Sufficient conditions for efficient indexing under different matchings. In: Proceedings of 30th annual symposium on combinatorial pattern matching (CPM 2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
2. Amir, A., Benson, G., Farach, M.: An alphabet independent approach to two-dimensional pattern matching. SIAM J. Comput. **23**(2), 313–323 (1994)
3. Berkman, O., Schieber, B., Vishkin, U.: Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. J. Algorithms **14**(3), 344–370 (1993)
4. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun of the ACM **20**(10), 762–772 (1977). https://doi.org/10.1145/359842.3598599
5. Cantone, D., Faro, S., Külekci, M.O.: An efficient skip-search approach to the order-preserving pattern matching problem. In: PSC, pp 22–35 (2015)
6. Chhabra, T., Tarhio, J.: A filtration method for order-preserving matching. Inf. Process. Lett. **116**(2), 71–74 (2016). https://doi.org/10.1016/j.ipl.2015.10.005
7. Chhabra, T., Külekci, M.O., Tarhio, J.: Alternative algorithms for order-preserving matching. In: PSC, pp 36–46 (2015)
8. Cho, S., Na, J.C., Park, K., et al.: A fast algorithm for order-preserving pattern matching. Inf. Process. Lett. **115**(2), 397–402 (2015)
9. Cole, R.: Parallel merge sort. SIAM J. Comput. **17**(4), 770–785 (1988)
10. Cole, R., Hazay, C., Lewenstein, M., et al.: Two-dimensional parameterized matching. ACM Trans. Algorithms **11**(2), 1–12 (2014). https://doi.org/10.1145/2650220
11. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., et al.: Order-preserving indexing. Theor. Comput. Sci. **638**, 122–135 (2016). https://doi.org/10.1016/j.tcs.2015.06.050
12. Faro, S., Külekci, M. O.: Efficient algorithms for the order preserving pattern matching problem. In: International Conference on Algorithmic Applications in Management, Springer, pp 185–196 (2016)
13. Faro, S., Lecroq, T.: The exact online string matching problem: a review of the most recent results. ACM Comput. Surv. (CSUR) **45**(2), 1–42 (2013)
14. Hasan, M.M., Islam, A.S., Rahman, M.S., et al.: Order preserving pattern matching revisited. Pattern Recogn. Lett. **55**, 15–21 (2015)
15. Horspool, R.N.: Practical fast searching in strings. Softw: Pract. Exp. **10**(6), 501–506 (1980). https://doi.org/10.1002/spe.4380100608
16. JáJá, J.: An Introduction to Parallel Algorithms, vol. 17. Addison-Wesley, Reading (1992)
17. Jargalsaikhan, D., Diptarama, Ueki, Y. et al: Duel and sweep algorithm for order-preserving pattern matching. In: SOFSEM 2018: theory and practice of computer science 44th international conference on current trends in theory and practice of computer science, Krems, Austria, January 29-February 2, Proceedings pp 624-635, (2018)
18. Jargalsaikhan, D., Hendrian, D., Yoshinaka, R. et al: Parallel duel-and-sweep algorithm for the order-preserving pattern matching. In: International conference on current trends in theory and practice of informatics, pp 211–222 (2020)
19. Jargalsaikhan, D., Hendrian, D., Yoshinaka, R. et al: Parallel algorithm for pattern matching problems under substring consistent equivalence relations. In: 33rd Annual symposium on combinatorial pattern matching, CPM 2022, June 27-29, 2022, Prague, Czech Republic, LIPIcs, vol 223. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp 28:1–28:21, (2022a) https://doi.org/10.4230/LIPIcs.CPM.2022.28
20. Jargalsaikhan, D., Hendrian, D., Yoshinaka, R. et al: Parallel algorithm for pattern matching problems under substring consistent equivalence relations. CoRR abs/2202.13284. (2022b) https://arxiv.org/abs/2202.13284, 2202.13284

21. Kim, J., Eades, P., Fleischer, R., et al.: Order-preserving matching. Theoret. Comput. Sci. **525**, 68–79 (2014)
22. Knuth, D.E., Morris, J.H., Jr., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977). https://doi.org/10.1137/0206024
23. Kubica, M., Kulczyński, T., Radoszewski, J., et al.: A linear time algorithm for consecutive permutation pattern matching. Inf. Process. Lett. **113**(12), 430–433 (2013)
24. Matsuoka, Y., Aoki, T., Inenaga, S., et al.: Generalized pattern matching and periodicity under substring consistent equivalence relations. Theoret. Comput. Sci. **656**, 225–233 (2016)
25. Ueki, Y., Narisawa, K., Shinohara, A.: A fast order-preserving matching with q-neighborhood filtration using SIMD instructions. In: SOFSEM (Student Research Forum Papers/Posters), pp 108–115 (2016)
26. Vishkin, U.: Optimal Parallel Pattern Matching in Strings. International Colloquium on Automata, Languages, and Programming, pp. 497–508. Springer, Berlin (1985)
27. Vishkin, U.: Deterministic sampling: a new technique for fast pattern matching. SIAM J. Comput. **20**(1), 22–40 (1991)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.