


Parallel Algorithm for Pattern Matching Problems Under Substring Consistent Equivalence Relations

Davaajav Jargalsaikhan ✉


Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Diptarama Hendrian ✉ 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Ryo Yoshinaka ✉ 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Ayumi Shinohara ✉ 

Graduate School of Information Sciences, Tohoku University, Sendai, Japan

Abstract

Given a text and a pattern over an alphabet, the pattern matching problem searches for all occurrences of the pattern in the text. An equivalence relation \approx is a substring consistent equivalence relation (SCER), if for two strings X and Y , $X \approx Y$ implies $|X| = |Y|$ and $X[i : j] \approx Y[i : j]$ for all $1 \leq i \leq j \leq |X|$. In this paper, we propose an efficient parallel algorithm for pattern matching under any SCER using the “duel-and-sweep” paradigm. For a pattern of length m and a text of length n , our algorithm runs in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w \cdot n \log^2 m)$ work, with $O(\tau_n^t + \xi_m^t \log^2 m)$ time and $O(\tau_n^w + \xi_m^w \cdot m \log^2 m)$ work preprocessing on the Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM), where τ_n^t , τ_n^w , ξ_m^t , and ξ_m^w are parameters dependent on SCERs, which are often linear in n and m , respectively.

2012 ACM Subject Classification Theory of computation \rightarrow Pattern matching

Keywords and phrases parallel algorithm, substring consistent equivalence relation, pattern matching

Digital Object Identifier 10.4230/LIPIcs.CPM.2022.28

Related Version *Full Version*: <https://arxiv.org/abs/2202.13284>

Funding *Diptarama Hendrian*: JSPS KAKENHI Grant Number JP19K20208

Ryo Yoshinaka: JSPS KAKENHI Grant Numbers JP18K11150 and JP20H05703

Ayumi Shinohara: JSPS KAKENHI Grant Number JP21K11745

1 Introduction

The string matching problem is fundamental and widely studied in computer science. Given a text and a pattern, the string matching problem searches for all substrings of the text that match the pattern. Many matching functions that are used in different string matching problems, including exact [15], parameterized [4], order-preserving [14, 16] and cartesian-tree [18] matchings, fall under the class of *substring consistent equivalence relations* (SCERs) [17]. An equivalence relation on strings is an SCER, if two strings X and Y match under the equivalence relation, then they have equal length and $X[i : j]$ matches $Y[i : j]$, for all $1 \leq i \leq j < |X|$. Matsuoka et al. [17] generalized the KMP algorithm [15] for pattern matching problems under SCERs. They also investigated periodicity properties of strings under SCERs. Kikuchi et al. [13] proposed algorithms to compute the shortest and longest cover arrays for a given string under any SCER. Hendrian [9] generalized Aho-Corasick algorithm for the dictionary matching under SCERs.



© Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara; licensed under Creative Commons License CC-BY 4.0

33rd Annual Symposium on Combinatorial Pattern Matching (CPM 2022).

Editors: Hideo Bannai and Jan Holub; Article No. 28; pp. 28:1–28:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Table 1** Summary of the encoding complexities for some SCER on P-CRCW PRAM.

| | τ_n^t | τ_n^w | ξ_m^t | ξ_m^w |
|----------------|-------------|---------------|-------------|---------------|
| Exact | $O(1)$ | $O(1)$ | $O(1)$ | $O(1)$ |
| Parameterized | $O(\log n)$ | $O(n \log n)$ | $O(1)$ | $O(1)$ |
| Cartesian-tree | $O(\log n)$ | $O(n \log n)$ | $O(\log m)$ | $O(m \log m)$ |

Vishkin proposed two algorithms for exact pattern matching, pattern matching by dueling [19] and pattern matching by sampling [20]. 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 as in, for instance, the KMP algorithm [15]. These algorithms are developed for parallel processing.

The dueling technique by Vishkin [19] has been proved to be useful for various kinds of pattern matching. Amir et al. [2] proposed a duel-and-sweep algorithm for two-dimensional exact matching, which is named “consistency and verification”. Cole et al. [8] extended it to two-dimensional parameterized matching. In addition, Jargalsaikhan et al. [11, 12] proposed serial and parallel duel-and-sweep algorithms for order-preserving matching.

In this paper, we propose an efficient parallel algorithm based on the dueling technique for the pattern matching problem under SCERs. Our parallel algorithm is the first to solve the problem under an arbitrary SCER in parallel. While Vishkin’s dueling algorithm for exact matching depends on the preferable properties of periods of strings, many of those do not hold with SCERs. Therefore, our algorithm involves new ideas and appears quite different from the original for exact pattern matching. For a pattern of length m and a text of length n , our algorithm runs in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w \cdot n \log^2 m)$ work, with $O(\tau_n^t + \xi_m^t \log^2 m)$ time and $O(\tau_n^w + \xi_m^w \cdot m \log^2 m)$ work preprocessing on the Priority Concurrent Read Concurrent Write Parallel Random-Access Machines (P-CRCW PRAM) [10]. Here, τ_n^t and τ_n^w are time and work respectively, needed on P-CRCW PRAM to encode in parallel a string X of length n under the SCER in concern. Given the encoding of X , ξ_m^t and ξ_m^w are time and work respectively to re-encode an element w.r.t. some suffix of X . Table 1 shows the encoding time and work complexities for some SCERs.

Due to the space restrictions, proofs of lemmas are relegated to Appendix B.

2 Preliminaries

We use Σ to denote an alphabet of symbols and 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. For a string $X \in \Sigma^*$ of length n , $X[i]$ denotes the i -th symbol of X , $X[i : j] = X[i]X[i+1] \dots X[j]$ denotes a substring of X that begins at position i and ends at position j for $1 \leq i \leq j \leq n$. For $i > j$, $X[i : j]$ denotes the empty string.

► **Definition 1** (Substring consistent equivalence relation (SCER) [17]). *An equivalence relation $\approx \subseteq \Sigma^* \times \Sigma^*$ is a substring consistent equivalence relation (SCER) if for two strings X and Y , $X \approx Y$ implies $|X| = |Y|$ and $X[i : j] \approx Y[i : j]$ for all $1 \leq i \leq j \leq |X|$.*

For instance, while the parameterized matching [4] and order-preserving matching [16, 14] are SCERs, the permutation matching [6, 7] and function matching [1] are not.

Hereafter we fix an arbitrary SCER \approx . We say that a position i is the *tight mismatch position* if $X[1 : i-1] \approx Y[1 : i-1]$ and $X[1 : i] \not\approx Y[1 : i]$. For two strings X and Y , let $LCP(X, Y)$ be the length l of the longest prefixes of X and Y match. That is, l is the

greatest integer such that $X[1 : l] \approx Y[1 : l]$. Obviously, if i is the tight mismatch position for $X \not\approx Y$, then $LCP(X, Y) = i - 1$. The converse holds if $i \leq \min\{|X|, |Y|\}$. Similarly, for a string X and an integer $0 \leq a < |X|$, we define $LCP_X(a) = LCP(X, X[a + 1 : |X|])$. In other words, $LCP_X(a)$ is the length of the longest common prefix, when X is superimposed on itself with offset a . We say $X \approx\text{-matches } Y$ iff $X \approx Y$. Given a text T of length n and a pattern P of length m , a position i in T , $1 \leq i \leq n - m + 1$, is an $\approx\text{-occurrence}$ of P in T iff $P \approx T[i : i + m - 1]$.

► **Definition 2** ($\approx\text{-pattern matching}$).

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

Output: All $\approx\text{-occurrences}$ of P inside T .

In the remainder of this paper, we fix text T to be of length n and pattern P to be of length m . We also assume that $n = 2m - 1$. Larger texts can be cut into overlapping pieces of length that are less than or equal to $(2m - 1)$ and processed independently. That is, we search for pattern occurrences in each substring $T[1 : 2m - 1], T[m + 1 : 3m - 1], \dots, T[\lfloor \frac{n-1}{m} \rfloor \cdot m + 1 : n]$, independently. For an integer x with $1 \leq x \leq n - m + 1$, a *candidate* T_x is the substring of T starting from x of length m , i.e., $T_x = T[x : x + m - 1]$.

For SCER matchings often it is convenient to encode the strings where $\approx\text{-equivalence}$ is reduced to the identity. Amir and Kondratovsky [3] showed that every SCER admits an encoding satisfying the following property.¹

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

► **Proposition 4.** An equivalence relation \approx is an SCER if and only if it admits an $\approx\text{-encoding}$.

Standard encodings of SCERs often satisfy the above definition, such as the prev-encoding [4] for parameterized matching and parent-distance encoding [18] for cartesian-tree matching. However, the nearest neighbor encoding [14] for order-preserving matching violates the third condition. Our algorithm for $\approx\text{-pattern matching}$ proposed in this paper relies on the property of Definition 3 and does not work with the nearest neighbor encoding. Nonetheless, duel-and-sweep algorithms for order-preserving matching based on the encoding are possible by further elaboration [11, 12], but we will not discuss it in this paper.

Fixing an $\approx\text{-encoding } f$, we denote $f(X)$ by \tilde{X} for simplicity. In addition, we denote the encoding of $X[x : |X|]$ as $\tilde{X}_x = f(X[x : |X|])$. Thus $\tilde{X}_1 = \tilde{X}$. For a string X , we suppose that \tilde{X} can be computed in $\tau_{|X|}^t$ time and $\tau_{|X|}^w$ work in parallel on P-CRCW PRAM. Moreover, we assume that given \tilde{X} , x , and i such that $x + i - 1 \leq |X|$, to compute $\tilde{X}_x[i]$, i.e. re-encoding the element at position i with respect to suffix $X[x : |X|]$, takes $\xi_{|X|}^t$ time and $\xi_{|X|}^w$ work on P-CRCW PRAM. Note that, to compute $\tilde{X}_x[i]$ does not necessarily require computing the whole of \tilde{X}_x . Those parameters are often reasonably small. See Table 1 and Appendix A for the prev-encoding for parameterized matching and the parent-distance encoding for cartesian-tree matching.

Vishkin's dueling technique essentially depends on the preferable properties of periods of strings. Matsuoka et al. [17] have discussed in detail how the classical notion of periods and their properties can be generalized when considering SCER matching. Unfortunately,

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

none of the generalizations yield a straightforward adaptation of Vishkin’s algorithm for SCER matching. Among those, the kind of periods involved in the duel-and-sweep algorithm discussed in this paper is *border-based period*.

► **Definition 5** (Border-based period). *Given a string X of length n , positive integer $p < n$ is called a border-based period of X if $X[1 : n - p] \approx X[p + 1 : n]$.*

Throughout the rest of the paper, we will refer to a border-based period as a *period*.

The family of models of computation used in this work is the priority concurrent-read concurrent-write (P-CRCW) PRAM [10]. This model allows simultaneous reading from the same memory location as well as simultaneous writing. In case of multiple writes to the same memory cell, the P-CRCW PRAM grants access to the memory cell to the processor with the smallest index.

3 Parallel algorithm for pattern matching under SCERs

We give an overview of the duel-and-sweep algorithm [2, 19]. The pattern is first preprocessed to obtain a *witness table*, which is later used to prune candidates during the pattern searching. As the name suggests, in the duel-and-sweep algorithm, the pattern searching is divided into two stages: the *dueling stage* and the *sweeping stage*. The pattern searching algorithm prunes candidates that cannot be pattern occurrences, first by performing “duels” between them, and then by “sweeping” through the remaining candidates to obtain pattern occurrences.

First, we explain the idea of dueling. Suppose P is superimposed on itself with an offset $a < m$ and the two overlapped regions of P do not match under \approx . Then it is impossible for two candidates T_x and T_{x+a} with offset a to match P simultaneously (see Figure 1). 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, it is guaranteed that if distinct candidates T_x and T_{x+a} that survive the dueling stage overlap, 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 advantage of this property when checking whether surviving candidates and the pattern match, so that this stage can also be done quickly.

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

More formally, in the dueling stage, we “duel” positions x and $x + a$ such that $\mathcal{W}_P(a) \neq \emptyset$ based on the following observation (see Figure 1).

► **Lemma 6.** *Suppose $w \in \mathcal{W}_P(a)$. Then,*

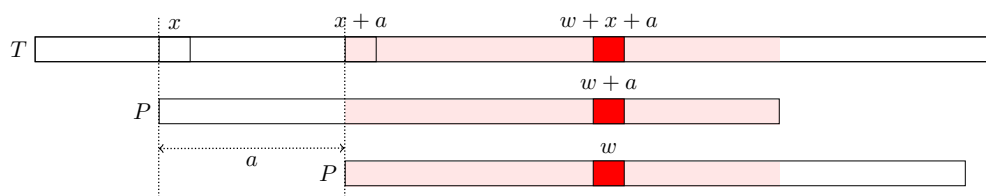
- *if $\tilde{T}_{x+a}[w] = \tilde{P}[w]$, then $T_x \not\approx P$,*
- *if $\tilde{T}_{x+a}[w] \neq \tilde{P}[w]$, then $T_{x+a} \not\approx P$.*

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

```

1 Function Dueling( $\tilde{S}, x, y$ )
2    $w \leftarrow W[y - x]$ ;
3   if  $\tilde{S}_y[w] = \tilde{P}[w]$  then return  $y$ ;
4   return  $x$ ;

```



■ **Figure 1** If $T_x \approx P \approx T_{x+a}$, then the overlapped regions of P superimposed on itself with offset a should match, i.e., $P_{a+1}[1 : m - a] \approx P[1 : m - a]$. If the overlapped region does not match, there must be a witness w such that $\tilde{P}_{a+1}[w] \neq \tilde{P}[w]$. Candidate positions x and $x + a$ perform a duel using the witness w based on Lemma 6.

Based on this lemma, we can safely eliminate either candidate T_x or T_{x+a} without looking into other positions. This process is called *dueling* (Algorithm 1). On the other hand, if the offset a has no witness, i.e. $P[1 : m - a] \approx P[a + 1 : m]$, no dueling is performed on them. We say that a position x is consistent with $x + a$ if $\mathcal{W}_P(a) = \emptyset$.

► **Lemma 7.** For any a, b, c such that $0 < a \leq b \leq c < m$, if a is consistent with b and b is consistent with c , then a is consistent with c .

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

Hereinafter, in our pseudo-codes we will use “ \leftarrow ” to note assignment operation into a local variable of a processor or assignment operation into a global variable which is accessed by a single processor at a time. 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.

3.1 Pattern preprocessing

The goal of the preprocessing stage is to compute a witness table $W[0 : m - 1]$, where $W[a] = 0$ if $\mathcal{W}_P(a) = \emptyset$, and $W[a] \in \mathcal{W}_P(a)$ otherwise. Algorithm 2 computes the tight mismatch position for X and Y , given \tilde{X} and \tilde{Y} .

► **Lemma 8.** For strings X and Y of equal length, given \tilde{X} and \tilde{Y} , Algorithm 2 computes the tight mismatch position in $O(1)$ time and $O(|X|)$ work on the P -CRCW PRAM.

■ **Algorithm 2** Check in parallel whether X and Y match, given \tilde{X} and \tilde{Y} . If they do not match, it returns the tight witness.

```

1 Function GetTightMismatchPos( $\tilde{X}, \tilde{Y}$ )
2    $w \leftarrow 0$ ;
3   for each  $i \in \{1, \dots, |X|\}$  do in parallel
4     if  $\tilde{X}[i] \neq \tilde{Y}[i]$  then  $w \leftarrow i$ ;
5   return  $w$ ;
```

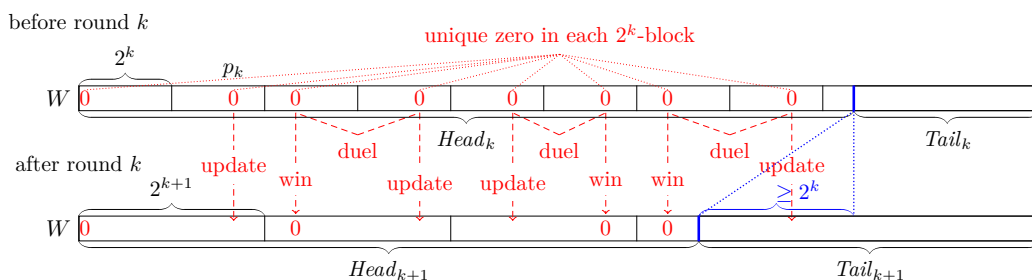
■ **Algorithm 3** Parallel algorithm for the pattern preprocessing.

```

1 Function PreprocessingParallel()
2    $tail \leftarrow m, k \leftarrow 0$ ;          /*  $tail$  is the starting position of  $Tail_k$  */
3   while  $2^k \leq tail$  do
4      $p \leftarrow \text{GetZeros}(2^k, 2^{k+1} - 1, k)[0]$ ;
5      $W[p] \leftarrow \text{GetTightMismatchPos}(\tilde{P}[1 : m - p], \tilde{P}_{p+1}[1 : m - p])$ ;
6     if  $W[p] = 0$  then  $lcp \leftarrow m - p$ ;
7     else  $lcp \leftarrow W[p] - 1$ ;
8      $old\_tail \leftarrow tail$ ;
9      $tail \leftarrow \min(old\_tail - 2^k, m - lcp)$ ;
10     $\text{SatisfySparsity}(tail - 1, k)$ ;
11     $\text{FinalizeTail}(tail, old\_tail, p, k)$ ;
12     $k \leftarrow k + 1$ ;
```

One can compute a witness table naively inputting $\tilde{P}[1 : m - a]$ and $\tilde{P}_{a+1}[1 : m - a]$ for all the offsets $a < m$ to Algorithm 2. However, this naive method costs as much as $\Omega(\xi_m^w \cdot m^2)$ work. We will present a more efficient algorithm in this subsection.

Our pattern preprocessing algorithm is described in Algorithm 3 and its outline is illustrated in Figure 2. Initially, all entries of the witness table are set to zero. Throughout preprocessing, each element of W is updated at most once. Therefore, at any point of the execution of the preprocessing algorithm, if $W[i] \neq 0$, then it must hold $W[i] \in \mathcal{W}_P(i)$. We say that position i is *finalized* if $W[i] = 0$ implies $\mathcal{W}_P(i) = \emptyset$ and $W[i] \neq 0$ implies $W[i] \in \mathcal{W}_P(i)$. During the execution of Algorithm 3, 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 the round k of the while loop by $Head_k$ and $Tail_k$, respectively. The variable $tail$ in Algorithm 3 represents the starting position of the tail, or equivalently, the length of the head. 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, 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 $i = 0, \dots, \lfloor h/2^k \rfloor - 1$ and $W[\lfloor h/2^k \rfloor \cdot 2^k : h - 1]$ where $h = |Head_k|$ 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 entry possibly except that the last 2^k -block has no zero entry. 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 ($W[0] = 0$) and $W[1 : 2^k - 1]$ contains no zeros.



■ **Figure 2** Illustration of the preprocessing invariant. W is partitioned into head and tail. The head is 2^k -sparse and the tail is finalized. The 2^k -sparsity is achieved by duels. The tail grows by at least 2^k at each round.

■ **Algorithm 4** 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  $A[0 : \lfloor r/2^k \rfloor - \lfloor l/2^k \rfloor]$  and initialize elements to  $-1$ ;
3   for each  $i \in \{l, \dots, r\}$  do in parallel
4     if  $W[i] = 0$  then  $A[\lfloor i/2^k \rfloor - \lfloor l/2^k \rfloor] \leftarrow i$ ;
5   return  $A$ ;

```

Initially, the entire table is the head and the size of the tail is zero: $Head_0 = W$ and $Tail_0 = \varepsilon$. The head is shrunk and the tail is extended by the following rule. Let the *suspected period* p_k at round k be the first zero position after the index 0, i.e., p_k is the unique position in the second 2^k -block such that $W[p_k] = 0$. Then, we let $Head_{k+1} = W[0 : m - x - 1]$ and $Tail_{k+1} = W[m - x : m - 1]$ for $x = |Tail_{k+1}| = \max(|Tail_k| + 2^k, LCP_P(p_k))$. When $|Head_k| < 2^k$, the 2^k -sparsity means that all the positions in the witness table are finalized. So, Algorithm 3 exits the while loop and halts. The goal of this subsection is to show the following theorem.

► **Theorem 9.** *Given \tilde{P} , the pattern preprocessing Algorithm 3 computes a witness table in $O(\xi_m^t \cdot \log^2 m)$ time and $O(\xi_m^w \cdot m \log^2 m)$ work on the P-CRCW PRAM.*

Proof. By Lemmas 13 and 15. ◀

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 4, which returns positions $i \in \{l, \dots, r\}$ such that $W[i] = 0$ in an array, assuming that $W[0 : r]$ satisfies the 2^k -sparsity. Algorithm 4 runs in $O(1)$ time and $O(r)$ work on the P-CRCW PRAM.

Comparison with Vishkin’s algorithm

The preprocessing algorithm for exact matching by Vishkin [19] also constructs a witness table so that it satisfies the 2^k -sparsity, incrementing k , where it has no head/tail separation. Maintaining the 2^k -sparsity for the whole table is possible due to the periodicity property which holds for the exact identity but not for general SCERs. Let $p \leq \lfloor i/2 \rfloor$ be the shortest period of $P[: i]$ for some i . In exact matching if $P[i] \neq P[i + j - 1]$, $P[i - p] \neq P[i + j - 1]$ holds. Thus, we can update $W[j + p]$ by using $W[j]$ i.e., we may let $W[j + p] = W[j] - p$. However, this property does not hold on SCERs generally. Still, Vishkin’s technique for

■ **Algorithm 5** Satisfy 2^{k+1} -sparsity of $Head_{k+1} = W[0 : x]$.

```

1 Function SatisfySparsity( $x, k$ )
2    $A \leftarrow \text{GetZeros}(2^{k+1}, x, k)$ ;
3   for each  $i \in \{0, 1, \dots, \lfloor |A|/2 - 1 \rfloor\}$  do in parallel
4      $j_1 \leftarrow A[2i], j_2 \leftarrow A[2i + 1]$ ;
5     if  $j_1 \neq -1$  and  $j_2 \neq -1$  then
6        $surv \leftarrow \text{Dueling}(\tilde{P}, j_1, j_2)$ ;
7        $a \leftarrow j_2 - j_1$ ;
8       if  $surv = j_1$  then  $W[j_2] \leftarrow W[a]$ ;
9       if  $surv = j_2$  then  $W[j_1] \leftarrow W[a] + a$ ;

```

keeping the 2^k -sparsity can partially be applied to SCER cases under a certain condition (Lemma 10), which requires us to control the length of the head part carefully. Concerning the tail part, where Vishkin's technique does not work, we design a new efficient algorithm for computing witnesses.

Head invariant

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 3, it satisfies the following invariant properties.

- $Head_k$ is 2^k -sparse.
- For all positions i of $Head_k$,
 - $W[i] \neq 0$ implies $W[i] \in \mathcal{W}_P(i)$,
 - $W[i] \leq |Tail_k| + 2^k$.

The head maintenance procedure **SatisfySparsity** is described in Algorithm 5. Before calling the function **SatisfySparsity**, Algorithm 3 finalizes the suspected period p_k , the first position after 0 such that $W[p_k] = 0$. Due to the 2^k -sparsity, $2^k \leq p_k < 2^{k+1}$. Algorithm 3 finds the suspected period p_k at Line 4 and then finalizes the position p_k at Line 5.

Let us explain how Algorithm 5 works. The task of **SatisfySparsity**(x, k) is to make $W[0 : x]$ 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 $tail = m - LCP_P(p_k) = p_k < 2^{k+1}$ when Algorithm 3 calls **SatisfySparsity**($tail - 1, k$). Then the array A obtained at Line 2 is empty and **SatisfySparsity**($tail - 1, k$) does nothing. After finalizing $Tail_{k+1}$, which will be explained later, the algorithm will halt without going into the next loop, since $|Head_{k+1}| \leq m - LCP_P(p_k) = p_k < 2^{k+1}$. At that moment all positions of W are finalized.

Hereafter we suppose that p_k is not a period of P . When **SatisfySparsity**($tail - 1, k$) is called, the value of $W[p_k]$ is the 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 one. The witness used for the duel between i and j is $W[a]$ for $a = j - i$, which is in the first 2^{k+1} -block. The following two lemmas ensure that indeed such duels are possible. Suppose that the pattern is superimposed on itself with offsets i and j . Lemma 10 below claims that if we already know $w \in \mathcal{W}_P(a)$ and $j + w \leq m$, in other words, if the witness lies within the overlap region, then we can obtain a witness for one of the offsets i and j by dueling them using w , without looking into other positions. Lemma 11 ensures that indeed we have a witness $w = W[a]$ in our table such that $j + w \leq m$ holds.

► **Lemma 10.** *For two offsets i and $j = i + a$ with $a > 0$, suppose $w \in \mathcal{W}_P(a)$ and $j + w \leq m$. Then,*

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

► **Lemma 11.** *For round k , suppose the preprocessing invariant holds true and $\mathcal{W}_P(p_k) \neq \emptyset$. Then, when SatisfySparsity is about to be called at Line 10 of Algorithm 3, for any two positions i and j of Head_{k+1} such that $0 < j - i < 2^{k+1}$, it holds that $j + W[j - i] \leq m$.*

Algorithm 5 updates the witness table in accordance with Lemma 10. In this way, the 2^k -sparsity of the head and the correctness of (non-zero) witnesses in the head are maintained. The invariants $W[i] \leq |\text{Tail}_k| + 2^k$ and $|\text{Head}_k| + \text{LCP}_P(p_k) \geq m$ are used in the proof of Lemma 11, which can be found in Appendix B.

► **Lemma 12.** *At the beginning of round k , for all $i \in \{0, \dots, 2^k - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 1$ and for all $i \in \{2^k, \dots, |\text{Head}_k| - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 2^k$.*

► **Lemma 13.** *In the round k of the while loop, Algorithm 5 updates the witness table so that Head_{k+1} is 2^{k+1} -sparse in $O(\xi_m^t)$ time and $O(\xi_m^w \cdot m/2^k)$ work on P-CRCW PRAM.*

Tail invariant

Next, we discuss how the algorithm finalizes Tail_{k+1} in the round k . This procedure is described in Algorithm 6. For the sake of convenience, we denote by \mathcal{T}_k the set of positions of Tail_k . Since Tail_k has already been finalized, it is enough to update $W[i]$ for $i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k$. We have two cases depending on how much the tail is extended.

The first case where $|\text{Tail}_{k+1}| = |\text{Tail}_k| + 2^k$ is handled naively. Since Head_k satisfies the 2^k -sparsity by the invariant, there are at most two zero positions in $\mathcal{T}_{k+1} \setminus \mathcal{T}_k$. Algorithm 6 naively uses Algorithm 2 to finalize those positions.

Now, we consider the case $|\text{Tail}_{k+1}| = \text{LCP}_P(p_k) > |\text{Tail}_k| + 2^k$. The following lemma is a key to handle this case.

► **Lemma 14.** *Suppose $m - \text{LCP}_P(p) \leq b < m$. If $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$ for any offset a such that $0 \leq a \leq b$ and $a \equiv b \pmod{p}$.*

Let us partition $\mathcal{T}_{k+1} \setminus \mathcal{T}_k$ into p_k subsets $\mathcal{S}_0, \dots, \mathcal{S}_{p_k-1}$ where $\mathcal{S}_s = \{i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k \mid i \equiv s \pmod{p_k}\}$, some of which can be empty. Lemma 14 implies that for each $s \in \{0, \dots, p_k - 1\}$, there exists a boundary offset b_s such that, for every $i \in \mathcal{S}_s$, $\mathcal{W}_P(i) = \emptyset$ iff $i > b_s$. Fortunately, for many s , one can find the boundary b_s very easily, unless $\mathcal{S}_s = \emptyset$. Let $q_s = \max \mathcal{S}_s$ for non-empty \mathcal{S}_s . Due to the 2^k -sparsity and the fact $p_k < 2^{k+1}$, it holds $W[q_s] \neq 0$ for all but at most three s . If $W[q_s] \neq 0$, then q_s is the boundary. By Lemma 14, $W[W[q_s] + q_s - i] \in \mathcal{W}_P(i)$ for all $i \in \mathcal{S}_s$. Accordingly, Algorithm 6 updates those values $W[i]$ in parallel in Lines 9–11.

On the other hand, for s such that $W[q_s] = 0$, Algorithm 7 uses binary search to find b_s and a witness $w \in \mathcal{W}_P(b_s)$ if it exists. Then, following Lemma 14, Algorithm 7 sets in parallel $W[i]$ to $w + (b_s - i)$ where $w \in \mathcal{W}_P(b_s)$ for $i \in \mathcal{S}_s$ such that $i \leq b_s$ (Line 10). If there is no boundary b_s in \mathcal{S}_s , then $\mathcal{W}_P(i) = \emptyset$ for all $i \in \mathcal{S}_s$. We do nothing in that case.

In Algorithm 7, the invariant is as follows. For $i \in \mathcal{S}_s$, $\mathcal{W}_P(i) \neq \emptyset$ if $i \leq l \cdot p_k + s$, and $\mathcal{W}_P(i) = \emptyset$ if $i \geq r \cdot p_k + s$. Each condition check of the binary search (Line 5) takes $O(\xi_m^t)$ time and $O(\xi_m^w \cdot m)$ work. Thus, the overall complexity of Algorithm 7 is $O(\xi_m^t \log m)$ time and $O(\xi_m^w \cdot m \log m)$ work.

► **Lemma 15.** *In round k , Algorithm 6 finalizes Tail_{k+1} in $O(\xi_m^t \log m)$ time and $O(\xi_m^w \cdot m \log m)$ work on P-CRCW PRAM.*

■ **Algorithm 6** Finalize $Tail_{k+1}$.

```

1 Function FinalizeTail(tail, old_tail, p, k)
2   if  $old\_tail - tail = 2^k$  then
3      $Z \leftarrow \text{GetZeros}(tail, old\_tail - 1, k);$  /*  $|Z| \leq 2$  */
4     for  $i = 0$  to  $|Z| - 1$  do
5        $z \leftarrow Z[i];$ 
6       if  $z \neq -1$  then
7          $W[z] \leftarrow \text{GetTightMismatchPos}(\tilde{P}[1:m-z], \tilde{P}_{z+1}[1:m-z])$ 
8     else
9       for each  $i \in \{tail, \dots, old\_tail - 1\}$  do in parallel
10         $q \leftarrow j$  where  $j \in \{old\_tail - p, \dots, old\_tail - 1\}$  and  $j \equiv i \pmod{p}$ ;
11        if  $W[i] = 0$  and  $W[q] \neq 0$  then  $W[i] \leftarrow W[q] + q - i;$ 
12       $Z \leftarrow \text{GetZeros}(old\_tail - p, old\_tail - 1, k);$  /*  $|Z| \leq 3$  */
13      for  $i = 0$  to  $|Z| - 1$  do
14         $z \leftarrow Z[i];$ 
15        if  $z \neq -1$  then Finalize(tail, old_tail, p,  $z \bmod p$ );

```

■ **Algorithm 7** Finalize $i \in \mathcal{T}_{k+1} \setminus \mathcal{T}_k$ s.t. $i \equiv s \pmod{p_k}$.

```

1 Function Finalize(tail, old_tail, p, s)
2    $l \leftarrow \lceil (tail - s)/p \rceil - 1, r \leftarrow \lfloor (old\_tail - 1 - s)/p \rfloor + 1;$ 
3   while  $r - l > 1$  do
4      $i \leftarrow \lfloor (l + r)/2 \rfloor, j \leftarrow i \cdot p + s;$ 
5     if  $\text{GetTightMismatchPos}(\tilde{P}[1:m-j], \tilde{P}_{j+1}[1:m-j]) = 0$  then  $r \leftarrow i;$ 
6     else  $l \leftarrow i;$ 
7    $b_s \leftarrow l \cdot p + s;$ 
8    $w \leftarrow \text{GetTightMismatchPos}(\tilde{P}[1:m-b_s], \tilde{P}_{b_s+1}[1:m-b_s]);$ 
9   for each  $i \in \{tail, \dots, b_s\}$  do in parallel
10    if  $W[i] = 0$  and  $i \equiv b_s \pmod{p}$  then  $W[i] \leftarrow w + b_s - i;$ 

```

3.2 Pattern searching

Our pattern searching algorithm prunes candidates in two stages: dueling and sweeping stages. During the dueling stage, candidate positions duel with each other, until the surviving candidate positions are pairwise consistent. 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 introduce a Boolean array $C[1:m]$ and initialize every entry of C to *True*. If a candidate T_i gets eliminated, we set $C[i] = \text{False}$. The pattern searching algorithm updates C in such a way that $C[i] = \text{True}$ iff i is a pattern occurrence. Entries of C are updated at most once during the dueling and sweeping stages.

Comparison with Vishkin's algorithm

When considering exact matching, Vishkin [19] found that if the pattern is periodic, i.e., $P = Q^j Q'$ for some aperiodic string Q , a proper prefix Q' of Q , and $j \geq 2$, the problem can be reduced to finding occurrences of Q and Q' in the text. Then a position i is an occurrence

■ **Algorithm 8** Parallel algorithm for the dueling stage.

```

1 Function DuelingStageParallel()
2   for each  $j \in \{1, \dots, m\}$  do in parallel
3      $\mathcal{C}_{0,j}[1] \leftarrow j$ ;
4    $k \leftarrow 1$ ;
5   while  $k \leq \lceil \log m \rceil$  do
6     for each  $j \in \{1, \dots, \lceil m/2^k \rceil\}$  do in parallel
7        $\mathcal{A} \leftarrow \mathcal{C}_{k-1,2j-1}$ ,  $\mathcal{B} \leftarrow \mathcal{C}_{k-1,2j}$ ;
8        $\langle a, b \rangle \leftarrow \text{Merge}(\mathcal{A}, \mathcal{B})$ ;
9       Let  $\mathcal{C}_{k,j}$  be array of length  $(a + |\mathcal{B}| - b + 1)$ ;
10      for each  $i \in \{1, \dots, a\}$  do in parallel
11         $\mathcal{C}_{k,j}[i] \leftarrow \mathcal{A}[i]$ ;
12      for each  $i \in \{b, \dots, |\mathcal{B}|\}$  do in parallel
13         $\mathcal{C}_{k,j}[a + i - b + 1] \leftarrow \mathcal{B}[i]$ ;
14     $k \leftarrow k + 1$ ;
15  Initialize all elements of  $C$  to False;
16  for each  $i \in \{1, \dots, \lceil C_{\lceil \log m \rceil, 1} \rceil\}$  do in parallel
17     $C[\lceil C_{\lceil \log m \rceil, 1} \rceil][i] \leftarrow \text{True}$ ;

```

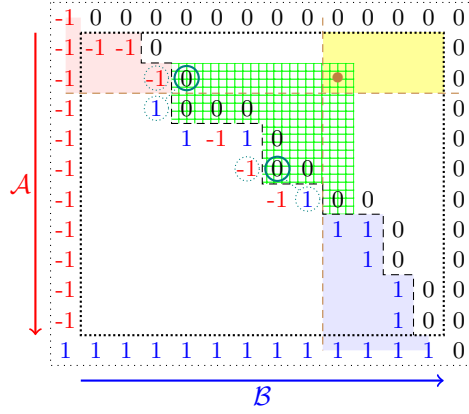
of P if and only if i is a starting position of j consecutive occurrences of Q followed by an occurrence of Q' . His dueling stage keeps the table C to be 2^k -sparse in the sense that $C[i] = \text{True}$ for at most one position i in every 2^k -block, incrementing k up to $\lfloor \log |Q|/2 \rfloor$. This can be done without violating the invariant since the occurrences of an aperiodic string Q are guaranteed to be sparse in the sense that the distance of two consecutive occurrences is bigger than $|Q|/2$. Then the sweeping stage naively checks whether those sparse surviving positions i with $C[i] = \text{True}$ are real occurrences. Apparently, this idea does not work at all in SCER matching. If P has a period p under an SCER, it does not mean that P is a repetition of $Q = P[1 : p]$ or that consecutive occurrences of Q form an occurrence of P . Our dueling and sweeping algorithms presented here are quite different from Vishkin's.

Dueling stage

The dueling stage is described in Algorithm 8. 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 $\mathcal{C}_{k,j} \subseteq \{(j-1)2^k + 1, \dots, j \cdot 2^k\}$ be the set of candidate positions in the j -th 2^k -block which have survived after the round k . The invariant of Algorithm 8 is as follows.

- At any point of execution of Algorithm 8, all pattern occurrences survive.
- For round k , each $\mathcal{C}_{k,j}$ is consistent.

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



■ **Figure 3** Padded grid G given two consistent sets \mathcal{A} and \mathcal{B} . All the occurrence positions are inside the yellow area, where the brown dot indicates \hat{i} and \hat{j} . The red- and blue-shaded areas consist of -1 and 1 only, respectively (Lemma 17). Any point (i, j) in the green-shaded area, where $i \geq \hat{i}$, $j \leq \hat{j}$, and $G[i][j] = 0$, is satisfactory to output. Among those, our algorithm finds a point (i, j) such that $G[i][j] = 0$, $G[i][j-1] = -1$ and $G[i+1][j'] = 1$ for some j' (Lemma 18). For example, the coordinates marked with green circles may be output.

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., $\max \mathcal{A} < \min \mathcal{B}$. Sets \mathcal{A} and \mathcal{B} should be merged in such a way that the resulting set is consistent and contains all occurrences in \mathcal{A} and \mathcal{B} . That is, we must find a consistent set \mathcal{C} such that $\hat{\mathcal{A}} \cup \hat{\mathcal{B}} \subseteq \mathcal{C} \subseteq \mathcal{A} \cup \mathcal{B}$, where $\hat{\mathcal{A}} = \{a \in \mathcal{A} \mid T_a \approx P\}$ and $\hat{\mathcal{B}} = \{b \in \mathcal{B} \mid T_b \approx P\}$ are the sets of occurrences in \mathcal{A} and \mathcal{B} , respectively. By the consistency property in Lemma 7, we can easily confirm that the following lemma holds.

► **Lemma 16.** *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 $\mathcal{B}_{\geq b} = \{j \in \mathcal{B} \mid j \geq b\}$.*

Therefore, it suffices to find $(a, b) \in \mathcal{A} \times \mathcal{B}$ such that a and b are consistent and $a \geq \max \hat{\mathcal{A}}$ and $b \leq \min \hat{\mathcal{B}}$. Then, $\mathcal{A}_{\leq a} \cup \mathcal{B}_{\geq b}$ has the desired property.

To find such a pair (a, b) , let us consider a grid G of size $(|\mathcal{A}| + 2) \times (|\mathcal{B}| + 2)$. 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 $\mathcal{B}[j]$, 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 technical convenience, we pad grid G with -1 s along the leftmost column, with 1 s along the bottom row, and with 0 s along the upper row and rightmost column. Specifically, $G[i][0] = -1$ for $i \in \{0, \dots, |\mathcal{A}|\}$, $G[|\mathcal{A}|+1][j] = 1$ for $j \in \{0, \dots, |\mathcal{B}|\}$, $G[i][|\mathcal{B}|+1] = 0$ for $i \in \{1, \dots, |\mathcal{A}|+1\}$, and $G[0][j] = 0$ for $j \in \{1, \dots, |\mathcal{B}|+1\}$. We will not compute the whole G , but this concept helps understanding the behavior of our algorithm. Figure 3 illustrates the grid, where elements of \mathcal{A} and \mathcal{B} are presented along the directions of rows and columns, respectively.

Lemma 16 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 separation line looks like a step function (Figure 3). In terms of the grid representation, our goal is to find a coordinate (i, j) in the zero region which is to the lower left of (\hat{i}, \hat{j}) (brown dot in Figure 3) where $\hat{i} = \max(\{i' \mid \mathcal{A}[i'] \in \hat{\mathcal{A}}\} \cup \{0\})$ and $\hat{j} = \min(\{j' \mid \mathcal{B}[j'] \in \hat{\mathcal{B}}\} \cup \{|\mathcal{B}| + 1\})$. Those points are

■ **Algorithm 9** Merge two consistent sets \mathcal{A} and \mathcal{B} .

```

1 Function Merge( $\mathcal{A}, \mathcal{B}$ )
2    $l_A \leftarrow 0, r_A \leftarrow |\mathcal{A}| + 1, j \leftarrow 1;$ 
3   while  $r_A - l_A > 1$  do
4      $m_A \leftarrow \lfloor (l_A + r_A)/2 \rfloor, \text{observedOne} \leftarrow \text{False};$ 
5      $l_B \leftarrow 0, r_B \leftarrow |\mathcal{B}| + 1;$ 
6     while  $r_B - l_B > 1$  do
7        $m_B \leftarrow \lfloor (l_B + r_B)/2 \rfloor;$ 
8       if  $W[\mathcal{B}[m_B] - \mathcal{A}[m_A]] = 0$  then  $r_B \leftarrow m_B;$ 
9       else
10        if Dueling( $\tilde{T}, \mathcal{A}[m_A], \mathcal{B}[m_B]$ ) =  $\mathcal{A}[m_A]$  then  $l_B \leftarrow m_B;$ 
11        else
12           $\text{observedOne} \leftarrow \text{True};$ 
13          break;
14    if  $\text{observedOne}$  then  $r_A \leftarrow m_A;$ 
15    else  $l_A \leftarrow m_A, j \leftarrow r_B;$ 
16  return  $\langle l_A, j \rangle;$ 

```

shown as the green-shaded area in Figure 3. Then, $\mathcal{A}_{\leq \mathcal{A}[i]} \cup \mathcal{B}_{\geq \mathcal{B}[j]}$ has the desired property. The region $\{(i', j') \mid i' \leq \hat{i} \text{ and } j' \geq \hat{j}\}$ is shaded with yellow, which consists of only zeros by Lemma 16.

The following lemma helps us to find a desired point.

► **Lemma 17.** *If $i \leq \hat{i}$, then row i consists only of non-positive elements. Similarly, if $j \geq \hat{j}$, then column j consists only of non-negative elements.*

Our algorithm seeks for a coordinate (i, j) in the following lemma.

► **Lemma 18.** *There always exists a pair (i, j) such that $i \leq |\mathcal{A}|$, $j \geq 1$, $G[i][j] = 0$, $G[i][j-1] = -1$ and $G[i+1][j'] = 1$ for some j' . For such (i, j) , it holds that $\hat{\mathcal{A}} \cup \hat{\mathcal{B}} \subseteq \mathcal{A}_{\leq \mathcal{A}[i]} \cup \mathcal{B}_{\geq \mathcal{B}[j]}$, assuming $\mathcal{A}_{\leq \mathcal{A}[0]} = \mathcal{B}_{\geq \mathcal{B}[|\mathcal{B}|+1]} = \emptyset$.*

Let us call a row i *low* if $G[i][j] = 0$ and $G[i][j-1] = -1$ for some j , and *high* if $G[i][j'] = 1$ for some j' . Note that, by Lemma 16 and the padding, each row is either low, high, or simultaneously low and high. At any point of Algorithm 9 execution, r_A is high and l_A is low, particularly $G[l_A][j] = 0$ and $G[l_A][j-1] = -1$. When $r_A = l_A + 1$, we are done by Lemma 18. In the inner while loop, we try to see whether the row $m_A = \lfloor (l_A + r_A)/2 \rfloor$ is high or low. As soon as we learn that m_A is high, we update r_A to be m_A . If m_A is revealed to be low, l_A is updated to be m_A .

► **Lemma 19.** *At any point of Algorithm 9 execution, (1) $G[l_A][j-1] = -1$ and $G[l_A][j] = 0$, (2) $G[r_A][j'] = 1$ for some j' , and (3) $G[m_A][l_B] = -1$ and $G[m_A][r_B] = 0$.*

► **Lemma 20.** *Given a witness table, \tilde{P} , and \tilde{T} , the dueling stage runs in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w m \log^2 m)$ work on P-CRCW-PRAM.*

■ **Algorithm 10** Parallel algorithm for the sweeping stage.

```

1 Function SweepingStageParallel()
2   create  $R[1 : m]$  and initialize elements of  $R$  to 0;
3    $k \leftarrow \lceil \log m \rceil$ ;
4   while  $k \geq 0$  do
5     create  $Piv[0 : \lfloor m/2^k \rfloor]$  and initialize its elements to  $-1$ ;
6     for each  $i \in \{1, \dots, m\}$  do in parallel
7       if  $C[i] = True$  and  $(i \bmod 2^k) > 2^{k-1}$  then  $Piv[\lfloor i/2^k \rfloor] \leftarrow i$ ;
8     for each  $b \in \{0, \dots, \lfloor m/2^k \rfloor\}$  do in parallel
9        $x \leftarrow Piv[b]$ ;
10      if  $x \neq -1$  then
11         $w \leftarrow \text{GetTightMismatchPos}(\tilde{P}[R[x] + 1 : m], \tilde{T}_x[R[x] + 1 : m])$ ;
12        if  $w = 0$  then  $R[x] \leftarrow m$ ;
13        else  $R[x] \leftarrow R[x] + w - 1$ ;
14      for each  $i \in \{1, \dots, m\}$  do in parallel
15         $x \leftarrow Piv[\lfloor i/2^k \rfloor]$ ;
16        if  $i \leq x$  and  $R[x] \leq m - (x - i) - 1$  then  $C[i] \leftarrow False$ ;
17        if  $i > x$  and  $C[i] = True$  then  $R[i] \leftarrow R[x] - (i - x)$ ;
18       $k \leftarrow k - 1$ ;

```

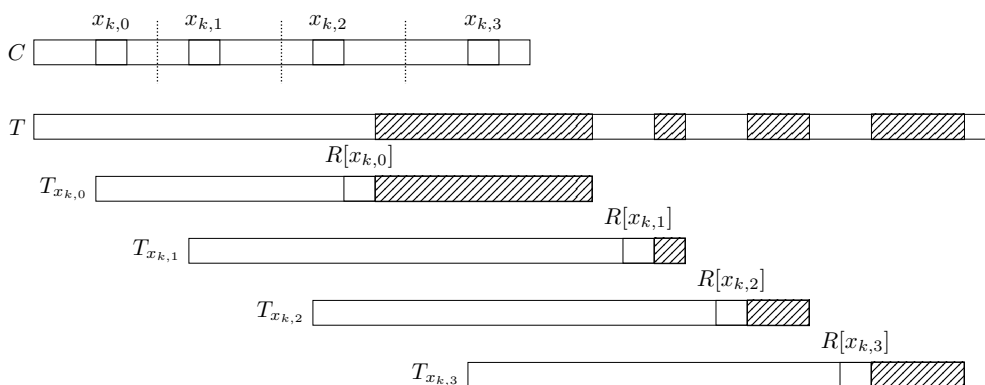
Sweeping stage

The sweeping stage is described in Algorithm 10. The sweeping stage updates C until $C[i] = True$ iff i is a pattern occurrence. All entries in C are updated at most once. Recall that all candidates that survived from the dueling stage are pairwise consistent. In addition to C , we will create a new integer array $R[1 : m]$. Throughout the sweeping stage, we have the following invariant properties:

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

The purpose of bookkeeping this information in R is to ensure that the sweeping stage algorithm uses $O(n)$ processors in each round. We do not want to access the same position of the text for each candidate covering the position. For two consistent candidate positions x and $x + a$ with $a > 0$, once we have calculated the value $r = LCP(T_x, P)$, we know that $LCP(T_{x+a}, P) \geq r - a$ for free, i.e., $\tilde{T}_{x+a}[1 : r - a] = \tilde{P}[1 : r - a]$. Then it suffices to check $\tilde{T}_{x+a}[r - a + 1 : m] = \tilde{P}[r - a + 1 : m]$. We keep the value $r - a$ in $R[x + a]$ for this trick, if $r - a \geq 0$. Throughout this section, we assume that a processor is attached to each position of C and T .

For each stage k , C is divided into 2^k -blocks. Unlike the preprocessing algorithm, k starts from $\lceil \log m \rceil$ and decreases with each round until $k = 0$. Let us look at each round in more detail. For the b -th 2^k -block of C , we pick as the “pivot” the smallest index $x_{k,b}$ in the second half of the 2^k -block such that $C[x_{k,b}] = True$. In Algorithm 10, we introduce array $Piv[0 : \lfloor m/2^k \rfloor]$ where $Piv[b] = x_{k,b}$. For each $x_{k,b}$, the algorithm computes $LCP(T_{x_{k,b}}, P)$ exactly and store the value in $R[x_{k,b}]$ on Lines 11–13. Suppose that $LCP(T_{x_{k,b}}, P) < m$, i.e., $T_{x_{k,b}} \not\approx P$ and $w = LCP(T_{x_{k,b}}, P) + 1$ is the tight mismatch position. Since all surviving candidate positions are pairwise consistent, if $T_{x_{k,b}} \not\approx P$, then, any candidate $T_{x_{k,b}-a}$ that “covers” w cannot match the pattern. Generally, we have the following.



■ **Figure 4** Illustrating the sweeping stage. The shaded regions of the text T are referenced during round k . Those referenced regions do not overlap.

► **Lemma 21.** *If two candidate positions x and $(x - a)$ with $a > 0$ are consistent and $LCP(T_x, P) \leq m - a - 1$, then $(x - a)$ is not an occurrence.*

Based on Lemma 21, Algorithm 10 updates $C[i]$ for indices i in the first half of each 2^k -block at Line 16. On the other hand, at Line 17, the algorithm updates the values of $R[i]$ for indices i in the second half of the block if $C[i] = True$. Since the surviving candidates are pairwise consistent, for candidate positions $(x_{k,b} + a)$ such that $a > 0$, $T_{x_{k,b}+a}[1 : r] \approx P[1 : r]$ for $r = R[x_{k,b}] - a$. In this way, the algorithm maintains the invariant properties. When $k = 0$, all the 2^k -blocks contain just one position x and $R[x]$ is set to be exactly $LCP(T_x, P)$ by Lines 11–13, unless $C[x] = False$ at that time. Then, if $R[x] < m$, then $C[x]$ will be *False* on Line 16. That is, when the algorithm halts, $C[x] = True$ iff $T_x \approx P$.

It remains to show the efficiency of the algorithm. We can prove the following lemmas.

► **Lemma 22.** *Each round of the while loop of Algorithm 10 can be performed in $O(\xi_m^t)$ time with $O(n)$ processors.*

► **Lemma 23.** *Given \tilde{P} and \tilde{T} , the sweeping stage algorithm finds all pattern occurrences in $O(\xi_m^t \log m)$ time and $O(\xi_m^w \cdot m \log m)$ work on the P-CRCW PRAM.*

By Theorem 9 and Lemmas 20 and 23, we obtain the main theorem. Recall that when $n \geq 2m$, T is cut into overlapping pieces of length $(2m - 1)$ and each piece is processed independently.

► **Theorem 24.** *Given a witness table, \tilde{P} , and \tilde{T} , the pattern searching solves the pattern searching problem under SCER in $O(\xi_m^t \cdot \log^3 m)$ time and $O(\xi_m^w \cdot n \log^2 m)$ work on the P-CRCW PRAM.*

4 Conclusion

Dueling [19] is a powerful technique, which enables us to perform pattern matching efficiently. In this paper, we have generalized the dueling technique for SCERs and have proposed a duel-and-sweep algorithm that solves the pattern matching problem for any SCER. Our algorithm is the first algorithm to solve any SCER pattern matching problem in parallel. Given a witness table, \tilde{P} , and \tilde{T} , we have shown that pattern searching under any SCER can be performed in $O(\xi_m^t \log^3 m)$ time and $O(\xi_m^w n \log^2 m)$ work on P-CRCW PRAM. Given

\tilde{P} , a witness table can be constructed in $O(\xi_m^t \log^2 m)$ time and $O(\xi_m^w \cdot m \log^2 m)$ work on P-CRCW PRAM. The third condition of \approx -encoding in Definition 3 ensures the generality of our duel-and-sweep algorithm for SCERs. However, some standard encoding method of an SCER, namely the nearest neighbor encoding for order-preserving matching, does not fulfill the third condition. We do not know if there is an alternative encoding for order-preserving matching that fulfills the condition and is computationally as cheap as the nearest neighbor encoding. Nevertheless, Jargalsaikhan et al. [11, 12] succeeded in designing a parallel duel-and-sweep algorithm for order-preserving matching using the nearest neighbor encoding, which appears quite similar to the SCER algorithm proposed in this paper. In our future work, we would like to investigate the relation between the encoding function and the dueling technique and further generalize the definition of encoding so that it becomes more inclusive.

References

- 1 Amihood Amir, Yonatan Aumann, Moshe Lewenstein, and Ely Porat. Function matching. *SIAM Journal on Computing*, 35(5):1007–1022, 2006.
- 2 Amihood Amir, Gary Benson, and Martin Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM Journal on Computing*, 23(2):313–323, 1994.
- 3 Amihood Amir and Eitan Konradovsky. 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.
- 4 Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of computer and system sciences*, 52(1):28–42, 1996.
- 5 Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.
- 6 Ayelet Butman, Revital Eres, and Gad M. Landau. Scaled and permuted string matching. *Information processing letters*, 92(6):293–297, 2004.
- 7 Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In *Proceedings of the Prague Stringology Conference 2009*, pages 105–117, 2009.
- 8 Richard Cole, Carmit Hazay, Moshe Lewenstein, and Dekel Tsur. Two-dimensional parameterized matching. *ACM Transactions on Algorithms (TALG)*, 11(2):12, 2014.
- 9 Diptarama Hendrian. Generalized dictionary matching under substring consistent equivalence relations. In *Proceedings of the 14th International Workshop on Algorithms and Computation*, pages 120–132, 2020.
- 10 Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992.
- 11 Davaajav Jargalsaikhan, Diptarama, Yohei Ueki, Ryo Yoshinaka, and Ayumi Shinohara. Duel and sweep algorithm for order-preserving pattern matching. In *Proceedings of the 44th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2018)*, pages 624–635, 2018.
- 12 Davaajav Jargalsaikhan, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Parallel duel-and-sweep algorithm for the order-preserving pattern matching. In *Proceedings of the 46th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2020)*, pages 211–222, 2020.
- 13 Natsumi Kikuchi, Diptarama Hendrian, Ryo Yoshinaka, and Ayumi Shinohara. Computing covers under substring consistent equivalence relations. In *Proceedings of the 27th International Symposium on String Processing and Information Retrieval*, pages 131–146, 2020.
- 14 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014.

- 15 Donald E. Knuth, James H. Morris, Jr, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.
- 16 Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.
- 17 Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theoretical Computer Science*, 656:225–233, 2016.
- 18 Sung Gwan Park, Amihod Amir, Gad M. Landau, and Kunsoo Park. Cartesian tree matching and indexing. In *Proceedings of the 30th Annual Symposium on Combinatorial Pattern Matching*, pages 16:1–16:14, 2019.
- 19 Uzi Vishkin. Optimal parallel pattern matching in strings. In *Proceedings of the 12th International Colloquium on Automata, Languages, and Programming*, pages 497–508, 1985.
- 20 Uzi Vishkin. Deterministic sampling — a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, 1991.

A Examples of encoding

Prev-encoding for parameterized matching

For a string X of length n over $\Sigma \cup \Pi$, where Π is an alphabet of parameter symbols and Σ is an alphabet of constant symbols, the *prev-encoding* [4] for X , denoted by $prev_X$, is defined to be a string over $\Sigma \cup \mathbb{N}$ of length n such that for each $1 \leq i \leq n$,

$$prev_X[i] = \begin{cases} X[i] & \text{if } X[i] \in \Sigma, \\ 0 & \text{if } X[i] \in \Pi \text{ and } X[i] \neq X[j] \text{ for } 1 \leq j < i, \\ i - k & \text{if } X[i] \in \Pi \text{ and } k = \max\{j \mid X[j] = X[i] \text{ and } 1 \leq j < i\}. \end{cases}$$

► **Theorem 25.** *Given a string X of length n , $prev_X$ can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM. Moreover, given $prev_X$, $prev_{X[x:n]}[i]$ can be computed in $O(1)$ time and $O(1)$ work.*

Proof. Without loss of generality, we assume that Π forms a totally ordered domain. We will construct the following string X' from X . We define a new symbol, say ∞ , such that, for any element $\pi \in \Pi$, π is less than ∞ . For $1 \leq i \leq |X|$, $X'[i] = X[i]$ if $X[i] \in \Pi$ and $X'[i] = \infty$ if $X[i] \in \Sigma$. For X' , we construct $Lmax_{X'}$, which is defined as $Lmax_{X'}[i] = j$ if $X'[j] = \max_{k < i} \{X'[k] \mid X'[k] \leq X'[i]\}$. We use the rightmost (largest) j if there exist more than one such $j < i$. If there is no such j , then we define $Lmax_{X'}[i] = 0$. Suppose that $X[i] \in \Pi$ for $1 \leq i \leq |X|$. After computing $Lmax_{X'}$, $prev_X[i] = i - Lmax_{X'}[i]$ if $X[i] = X[Lmax_{X'}[i]]$. If $Lmax_{X'}[i] = 0$ or $X[i] \neq X[Lmax_{X'}[i]]$, then $X[i]$ is the first occurrence of this letter. Thus, $prev_X$ can be computed from $Lmax_{X'}$ in $O(1)$ time and $O(n)$ work. Since $Lmax_{X'}$ can be computed in $O(\log n)$ time and $O(n \log n)$ work [12], overall complexities are $O(\log n)$ time and $O(n \log n)$ work.

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

$$prev_{X[x:n]}[i] = \begin{cases} 0 & \text{if } X[x+i-1] \in \Pi \text{ and } prev_X[x+i-1] \geq i, \\ prev_X[x+i-1] & \text{otherwise.} \end{cases} \blacktriangleleft$$

Parent-distance encoding for cartesian-tree matching

For a string X over a totally ordered alphabet, its parent-distance encoding [18] for cartesian-tree matching PD_X is defined as follows.

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

► **Theorem 26.** *Given a string X of length n , PD_X can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM. Moreover, given PD_X , $PD_{X[x:n]}[i]$ can be computed in $O(1)$ time and $O(1)$ work.*

Proof. For $1 \leq i \leq n$, $PD_X[i]$ is the nearest smaller value to the left of $X[i]$. Since the all-smaller-nearest-value problem can be solved in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM by Berkman et al. [5], PD_X can be computed in $O(\log n)$ time and $O(n \log n)$ work on P-CRCW PRAM.

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

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

B Proofs

► **Proposition 4.** *An equivalence relation \approx is an SCER if and only if it admits an \approx -encoding.*

Proof. It suffices to show the “if” direction. Suppose we have an \approx -encoding f . If $X \approx Y$, then $f(X) = f(Y)$ by (4) of Definition 3. In this case, we have $f(X[j:k])[i] = f(Y[j:k])[i]$ for any $1 \leq j \leq k \leq |X|$ and $1 \leq i \leq k-j+1$ by $f(X)[i+j-1] = f(Y)[i+j-1]$, (3), and (2). Hence, $X[j:k] \approx Y[j:k]$ by (4). ◀

► **Lemma 6.** *Suppose $w \in \mathcal{W}_P(a)$. Then,*

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

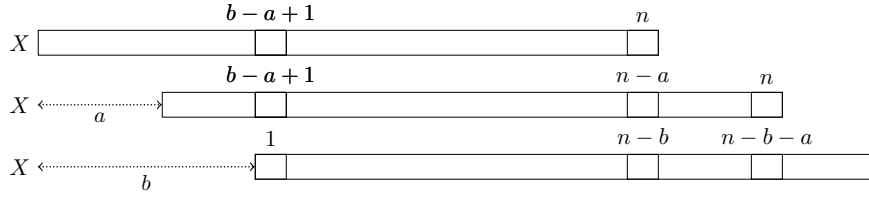
Proof. If $\tilde{T}_{x+a}[w] \neq \tilde{P}[w]$, then by the fourth property of the \approx -encoding (Definition 3), $T_{x+a} \not\approx P$. If $\tilde{T}_{x+a}[w] = \tilde{P}[w] \neq \tilde{P}_{a+1}[w]$, then by the third property of the \approx -encoding, $\tilde{T}_x[w+a] \neq \tilde{P}[w+a]$, so $T_x \not\approx P$. ◀

► **Lemma 27.** *Suppose that a and b are periods of X . If $a+b < |X|$, then $(b+a)$ is a period of X . If $a < b$, then $(b-a)$ is a period of $X[1:|X|-a]$.*

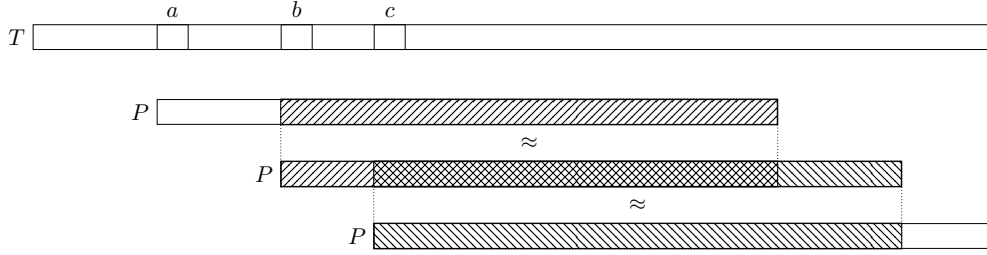
This lemma implies that if p is a period of X , then so is qp for every positive integer $q \leq \lfloor (|X|-1)/p \rfloor$.

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

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



■ **Figure 5** Suppose that a and b are periods of X . If $a + b < |X|$, then $(b + a)$ is a period of X . If $a < b$, then $(b - a)$ is a period of $X[1 : |X| - a]$.



■ **Figure 6** For candidate positions $a < b < c$, if a is consistent with b and b is consistent with c , then a is consistent with c .

► **Lemma 7.** For any a, b, c such that $0 < a \leq b \leq c < m$, if a is consistent with b and b is consistent with c , then a is consistent with c .

Proof. By Lemma 27. ◀

► **Lemma 11.** For round k , suppose the preprocessing invariant holds true and $\mathcal{W}_P(p_k) \neq \emptyset$. Then, when `SatisfySparsity` is about to be called at Line 10 of Algorithm 3, for any two positions i and j of Head_{k+1} such that $0 < j - i < 2^{k+1}$, it holds that $j + W[j - i] \leq m$.

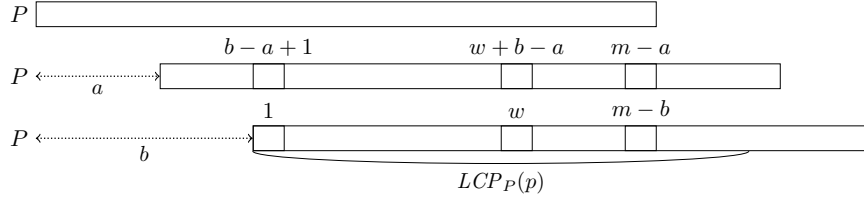
Proof. Let $a = j - i$ and $w = W[a]$. Recall that a belongs to the first 2^{k+1} -block and $W[a]$ is updated only if $a = p_k$. Suppose $a \neq p_k$. At the beginning of round k , by the invariant property, we have $w \leq |\text{Tail}_k| + 2^k$. Since $j < |\text{Head}_{k+1}| = m - |\text{Tail}_{k+1}|$,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

If $a = p_k$, $w = W[p_k]$ is the tight witness for offset p_k , i.e., $w = \text{LCP}_P(p_k) + 1$. Since $|\text{Tail}_{k+1}| \geq \text{LCP}_P(p_k)$, $j + w \leq j + |\text{Tail}_{k+1}| + 1$. Since $j < |\text{Head}_{k+1}|$, $j + |\text{Tail}_{k+1}| + 1 \leq |\text{Head}_{k+1}| + |\text{Tail}_{k+1}| \leq m$. We have proved that $j + w \leq m$. ◀

► **Lemma 12.** At the beginning of round k , for all $i \in \{0, \dots, 2^k - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 1$ and for all $i \in \{2^k, \dots, |\text{Head}_k| - 1\}$, it holds $W[i] \leq |\text{Tail}_k| + 2^k$.

Proof. We show the lemma by induction on k . At the beginning of round 0, every element of W is zero and $|\text{Tail}_0| = 0$, thus, the claim holds. We will show that the lemma holds for $k + 1$ assuming that it is the case for k .

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



■ **Figure 7** For offsets a, b such that $m - LCP_P(p) \leq b < m$ and $a \equiv b \pmod{p}$, if $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$.

Suppose $2^{k+1} \leq i < |Head_{k+1}|$. If Algorithm 5 does not update $W[i]$, by the induction hypothesis, $W[i] \leq |Tail_k| + 2^k < |Tail_{k+1}| + 2^{k+1}$ holds. Suppose Algorithm 5 updates $W[i]$ or $W[j]$ by a duel between i and j , where $2^{k+1} \leq i < j < |Head_{k+1}|$ and $a = j - i < 2^{k+1}$. We have shown above that $W[a] \leq |Tail_{k+1}| + 1$. If i wins the duel, then $W[j] = W[a] \leq |Tail_{k+1}| + 1 \leq |Tail_{k+1}| + 2^{k+1}$. If j wins the duel, then $W[i] = W[a] + a \leq |Tail_{k+1}| + 1 + a \leq |Tail_{k+1}| + 2^{k+1}$. ◀

► **Lemma 14.** *Suppose $m - LCP_P(p) \leq b < m$. If $w \in \mathcal{W}_P(b)$, then $(w + b - a) \in \mathcal{W}_P(a)$ for any offset a such that $0 \leq a \leq b$ and $a \equiv b \pmod{p}$.*

Proof. Figure 7 may help understanding the proof. Suppose $w \in \mathcal{W}_P(b)$, i.e., $\tilde{P}_{b+1}[w] \neq \tilde{P}[w]$. Since p is a period of $P[1 : LCP_P(p)]$ and $a \equiv b \pmod{p}$, by Lemma 27, $(b - a)$ is also a period of $P[1 : LCP_P(p)]$, i.e., $P[1 + b - a : LCP_P(p)] \approx P[1 : LCP_P(p) - (b - a)]$. Particularly for the position $w \leq m - b \leq m - a$, we have $\tilde{P}_{b-a+1}[w] = \tilde{P}[w]$. Then, $\tilde{P}_{b-a+1}[w] \neq \tilde{P}_{b+1}[w]$ by the assumption (Figure 7). By Property (3) of the \approx -encoding (Definition 3), $\tilde{P}_1[b - a + w] \neq \tilde{P}_{a+1}[b - a + w]$. That is, $(w + b - a) \in \mathcal{W}_P(a)$. ◀

► **Theorem 9.** *Given \tilde{P} , the pattern preprocessing Algorithm 3 computes a witness table in $O(\xi_m^t \cdot \log^2 m)$ time and $O(\xi_m^w \cdot m \log^2 m)$ work on the P-CRCW PRAM.*

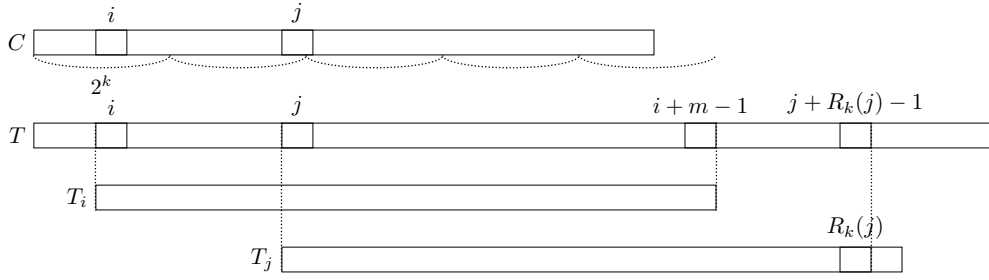
Proof. When the algorithm halts, by $2^k \leq tail$, the head size is at most 2^k . Therefore, the head is zero-free except for $W[0] = 0$ by the 2^k -sparsity. By the invariant, $W[i] \in \mathcal{W}_P(i)$ for all the positions of the head. On the other hand, every position of the tail is finalized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

In Algorithm 3, the while loop runs $O(\log m)$ times, and each loop takes $O(\xi_m^t \log m)$ time and $O(\xi_m^t \cdot m \log m)$ work, by Lemmas 13 and 15. Thus, the overall complexity of Algorithm 3 is $O(\xi_m^t \cdot \log^2 m)$ time and $O(\xi_m^t \cdot m \log^2 m)$ work. ◀

► **Lemma 17.** *If $i \leq \hat{i}$, then row i consists only of non-positive elements. Similarly, if $j \geq \hat{j}$, 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. Let us consider $i = \hat{i}$. Since \hat{i} is a pattern occurrence, for any j that \hat{i} is not consistent with, \hat{i} always wins the duel. Thus, if $G[\hat{i}][j] \neq 0$, then $G[\hat{i}][j] = -1$. Now, let us consider $i < \hat{i}$. For any $j \in \mathcal{B}$, $i < \hat{i} < j$. Since \mathcal{A} is a consistent set and \hat{i} is a pattern occurrence, if i is not consistent with j , i always wins any duel against j . Thus, if $G[i][j] \neq 0$, then $G[i][j] = -1$. ◀

► **Lemma 18.** *There always exists a pair (i, j) such that $i \leq |\mathcal{A}|$, $j \geq 1$, $G[i][j] = 0$, $G[i][j - 1] = -1$ and $G[i + 1][j'] = 1$ for some j' . For such (i, j) , it holds that $\hat{\mathcal{A}} \cup \hat{\mathcal{B}} \subseteq \mathcal{A}_{\leq \mathcal{A}[i]} \cup \mathcal{B}_{\geq \mathcal{B}[j]}$, assuming $\mathcal{A}_{\leq \mathcal{A}[0]} = \mathcal{B}_{\geq \mathcal{B}[|\mathcal{B}|+1]} = \emptyset$.*



■ **Figure 8** Before round k , for two surviving candidates T_i and T_j such that $j - i \geq 2^k$, $i + m - 1 < j + R_k[j]$.

Proof. Let $i = \max\{i' \mid G[i'][j'] \leq 0 \text{ for all } j'\}$ and $j = \min\{j' \mid G[i][j'] = 0\}$. It is easy to see that those are well-defined and satisfy the desired condition.

Suppose (i, j) satisfies the condition. Since $G[i'][j'] \leq 0$ for all $i \leq \hat{i}$ and j' by Lemma 17, $G[i+1][j'] = 1$ means that $i+1 > \hat{i}$, i.e., $i \geq \hat{i}$. Similarly, $G[i][j-1] = -1$ means $j \leq \hat{j}$. ◀

► **Lemma 21.** *If two candidate positions x and $(x - a)$ with $a > 0$ are consistent and $LCP(T_x, P) \leq m - a - 1$, then $(x - a)$ is not an occurrence.*

Proof. Let $w = LCP(T_x, P) + 1$. Then $w \leq m - a$ and $T_x[1 : m - a] \not\approx P[1 : m - a]$. Since x is consistent with $(x - a)$, $P[a + 1 : m] \approx P[1 : m - a] \not\approx T_x[1 : m - a] \approx T_{x-a}[a + 1 : m]$, which means that $(x - a)$ is not a pattern occurrence. ◀

Lemma 22 follows from Lemma 28.

► **Lemma 28.** *After the round k , for two surviving candidate positions i and j with $i < j$ that do not belong to the same 2^{k-1} -block of C , $i + m \leq j + R[j]$.*

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

Now, let us consider the case when candidate positions i and j belong to the same 2^k -block of C . During round k , for each 2^k -block of C , Algorithm 10 chooses as surviving candidate position $x_{k,b}$ which is the smallest index in the second half of the 2^k -block. Thus, two surviving candidate positions i and j of the b -th 2^k -block belong to different 2^{k-1} -blocks iff $i < x_{k,b} \leq j$. For T_i to be a surviving candidate after round k , it must be the case that $m + i \leq LCP(T_{x_{k,b}}, P) + x_{k,b}$. For T_j , Algorithm 10 updates $R_k[j]$ to $LCP(T_{x_{k,b}}, P) - (j - x_{k,b})$. Substituting it into the previous inequality, we get $m + i \leq R_k[j] + (j - x_{k,b}) + x_{k,b} = R_k[j] + j$. ◀

► **Lemma 22.** *Each round of the while loop of Algorithm 10 can be performed in $O(\xi_m^t)$ time with $O(n)$ processors.*

Proof. Obviously it runs in constant time except for the computation at Line 11, where each processor attached to position i is used for re-encoding $\tilde{T}[i]$ into $\tilde{T}_x[i - x + 1]$ and comparing the value with $\tilde{P}[i - x + 1]$ for some x . Indeed, there is at most one b such that $x_{k,b} + R[x_{k,b}] \leq i < x_{k,b} + m$, since $x_{k,b-1} + m \leq x_{k,b} + R[x_{k,b}]$ for all $b \in \{1, \dots, \lceil m/2^k \rceil\}$ by Lemma 28. ◀