Solving the Minimal Positional Substring Cover Problem in Sublinear Space

Paola Bonizzoni
Department of Computer Science, University of Milano-Bicocca, Italy

Christina Boucher
Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA

Davide Cozzi
Department of Computer Science, University of Milano-Bicocca, Italy

Travis Gagie
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Yuri Pirola
Department of Computer Science, University of Milano-Bicocca, Italy

Abstract

Within the field of haplotype analysis, the Positional Burrows-Wheeler Transform (PBWT) stands out as a key innovation, addressing numerous challenges in genomics. For example, Sanaullah et al. introduced a PBWT-based method that addresses the haplotype threading problem, which involves representing a query haplotype through a minimal set of substrings. To solve this problem using the PBWT data structure, they formulate the Minimal Positional Substring Cover (MPSC) problem, and then, subsequently present a solution for it. Additionally, they present and solve several variants of this problem: k-MPSC, leftmost MPSC, rightmost MPSC, and length-maximal MPSC. Yet, a full PBWT is required for each of their solutions, which yields a significant memory usage requirement. Here, we take advantage of the latest results on run-length encoding the PBWT, to solve the MPSC in a sublinear amount of space. Our methods involve demonstrating that k-Set Maximal Exact Matches (k-SMEMs) can be computed in a sublinear amount of space via efficient computation of k-Matching Statistics (k-MS). This leads to a solution that requires sublinear space for, not only the MPSC problem, but for all its variations proposed by Sanaullah et al. Most importantly, we present experimental results on haplotype panels from the 1000 Genomes Project data that show the utility of these theoretical results. We conclusively demonstrate that our approach markedly decreases the memory required to solve the MPSC problem, achieving a reduction of at least two orders of magnitude compared to the method proposed by Sanaullah et al. This efficiency allows us to solve the problem on large versions of the problem, where other methods are unable to scale to. In summary, the creation of μ-PBWT paves the way for new possibilities in conducting in-depth genetic research and analysis on a large scale. All source code is publicly available at `https://github.com/dlcgold/muPBWT/tree/k-smem`.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Positional Burrows–Wheeler Transform, r-index, minimal position substring cover, set-maximal exact matches

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.12

Supplementary Material *Software (Source code)*: `https://github.com/dlcgold/muPBWT/tree/k-smem` [2], archived at `swh:1:dir:d3467768a54423c8294abfc44f87f18705b3ed02`

Funding PB has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement ALPACA No. 956229. PB, DC, and YP have received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 872539. PB, DC,

and YP are also supported by the grant MIUR 2022YRB97K, PINC, Pangenome Informatics: from Theory to Applications. PB, DC, and YP are partially supported by the MUR under the grant "Dipartimenti di Eccellenza 2023-2027" of the Department of Informatics, Systems and Communication of the University of Milano-Bicocca, Italy. TG is funded by NSERC Discovery Grant RGPIN-07185-2020. CB is funded by NSF IIBR (Grant No. P0177268), NSF SCH (Grant No. P0212789) and NIH NIAID (Grant No. R01AI141810).

1 Introduction

In recent years, the Positional Burrows-Wheeler Transform (PBWT) has emerged as a fundamental data structure for solving various challenges in haplotype analysis. One challenge in haplotype analysis is *haplotype threading*, which aims to represent a query haplotype by using one or more substrings derived from at least k haplotypes within a reference panel. Sanaullah et al. [10, 11] showed that solutions to the Minimal Positional Substring Cover (MPSC) problem can be used to solve the *haplotype threading* problem in the context of the PBWT. This insight led to a PBWT-framework for solving haplotype threading that is an alternative to the classical Li and Stephens [7] model that is promised to be more scalable – as the original is unlikely to be efficient enough to be applied on large biobank datasets. In addition to the formulation and solution to the MPSC problem, Sanaullah et al. define several variants of this problem as well as algorithmic solutions. Although more space-efficient than the Li and Stephens model, each of the solutions presented by Sanaullah et al. require the construction and storage of the full PBWT. As it was previously mentioned by Durbin [4], storing the entire PBWT will scale linearly in the size of the input – more precisely, Durbin predicted it to be $13n$ bytes. For reasonably large biobank datasets – such as the ones offered by the 1000 Genomes Project data – this would quickly become unwieldy.

In this paper, we present a space- and time- efficient manner for solving MPSC as well as all variants suggested by Sanaullah et al: k-MPSC, leftmost MPSC, rightmost MPSC, and length-maximal MPSC. Our methods take advantage of the latest advancements in run-length encoding the PBWT, and exploit the fact that MPSC problem can be construed as computing all Set Maximal Exact Matches (SMEMs) in the PBWT, which are maximal matches that are common between a pattern and a panel that cannot be extended. More specifically, we show that we can efficiently compute k-Set Maximal Exact Matches (k-SMEMs), which have the additional constraint that any SMEM must occur in at least k rows in the reference panel. This efficient computation comes from extending the definition of Matching Statistics in the PBWT [3] to k-Matching Statistics (k-MS) in the PBWT. This allows for algorithms developed by Cozzi et al. [3] to compute Matching Statistics to be extended to computing k-MS, which yield k-SMEMs, and finally, a solution to the MPSC and the k-MPSC problem, i.e., find a MPSC in which each positional substring is covered by at least k rows in the reference panel. Hence, we show how k-SMEMs and k-MS naturally provide a framework for developing efficient solutions to the MPSC problem and its variants, proving that we can solve these problems in sublinear space with respect to the size of the panel.

Most importantly, we implement our approach and show that the sublinear bound on the space usage has a practical benefit in haplotype analysis. We compare our implementation to the current state-of-the-art on increasingly-larger autosome panels of the 1000 Genomes Project. Our findings on these datasets demonstrate that our methods substantially lowers memory consumption for solving the k-MPSC problem, achieving reductions of at least two orders of magnitude compared to the method presented by Sanaullah et al. Due to this memory usage, only our method was able to scale to the largest panel sizes in 1000 Genomes Project with reasonable memory constraints. Hence, we show our strategy is applicable of solving haplotype threading on extensive datasets found in modern biobanks.

2 Background

In this paper, we propose novel algorithms for solving the MPSC problem and its variants. Our algorithms are based on μ-PBWT, which is a run-length encoding of the PBWT. To lay the groundwork for describing the algorithms in this paper, we formally introduce the MPSC problems and its variants, and then introduce μ-PBWT and the concepts needed for the development of our methods.

2.1 The Minimal Positional Substring Cover Problems

Throughout this paper, we define a string X over a finite, ordered alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$ to be the concatenation of $|X| = n$ characters $X = X[1..n]$ of Σ. We denote the empty string as ε, the string spanning position i through j as $X[i..j]$ (with $X[i..j] = \varepsilon$ if $i > j$), the i-th prefix of X as $X[1..i]$, and the i-th suffix as $X[i..|X|]$.

A *positional substring* of a string X is a triplet (i, j, X) with $1 \leq i, j \leq |X|$ and we say that the substring corresponding to (i, j, X) is $X[i..j]$. Two positional substrings (i, j, X) and (k, l, Y) are *equal* iff $i = k$, $j = l$, and $X[i..j] = Y[k..l]$. A positional substring (i, j, X) is *contained* in a string Y iff $X[i..j] = Y[i..j]$.

Given a set S of strings of length w (i.e., panel), a *k-positional substring cover* of a w-length string P by S is a set C of positional substrings such that: *(i)* each position $l \in [1, w]$ of P is covered by a $(i, j, X) \in C$ (i.e., $i \leq l \leq j$), *(ii)* each $(i, j, X) \in C$ is contained in P, and *(iii)* each $(i, j, X) \in C$ is contained in at least k distinct strings of S. The size of the cover is the number of elements in C, which we denote as $|C|$.

▶ **Problem 1** (k-Minimal Positional Substring Cover problem, k-MPSC [10]). *Given a set S of h strings of length w and a string P of length w, find, if it exists, a k-positional substring cover of P by S with the smallest size over all k-positional substring covers of P by S.*

The MPSC problem is the k-MPSC problem where k is equal to 1 (i.e., each positional substring of the cover is contained in at least one string of the panel). It is easy to see that a solution to the problem exists iff for every i, with $1 \leq i \leq w$, the positional substrings (i, i, P) are contained in at least k distinct strings of S.

The best known algorithm for computing a k-MPSC requires $O(w)$ time [11] and it works column-wise from left to right by extending matches of the string P with at least k strings of the panel S. At any column, if the current match cannot be extended, then a new match starting at the current column is initiated. Optimality is ensured by the property of k-MPSC modularity [10, Lemma 2]. Matches of P with at least h strings in S are efficiently computed and extended using the Positional Burrows–Wheeler Transform (PBWT) of S. Hereon, we denote $h \cdot w$ as n, which will be used throughout this paper to bound the space and time complexity. The k-MPSC algorithm of Sanaullah et al. [10] requires $\mathcal{O}(n)$ space to ensure constant-time random access to the input panel and to the PBWT.

Next, we draw a relationship between positional substrings and a generalization of maximal exact matches, which we refer to as *k-Set Maximal Exact Match* (k-SMEM), by requiring the maximality of matches to be defined as follows.

▶ **Definition 2** (k-SMEM). *Let S be a set of h sequences of length w and let P be a string of length w. The pair (i, j) is a k-SMEM if the positional substring (i, j, P) is contained in at least k sequences of S and one of the following holds:*

- *$i = 1$ and $j = w$*
- *$i = 1$ and $(i, j + 1, P)$ is not contained in at least k strings of S*
- *$j = w$ and $(i - 1, j, P)$ is not contained in at least k strings of S*
- *$(i - 1, j, P)$ and $(i, j + 1, P)$ are not contained in at least k strings of S*

It is straightforward to observe that every positional substring (i, j, P), which is part of a k-MPSC of P by S, generates an interval that fits within a k-SMEM (i', j'), where $i' \leq i$ and $j' \geq j$. This is because (i, j) either directly constitutes a k-SMEM or can be extended either to the left, the right, or both.

▶ **Problem 3** (k-SMEM-finding). *Given a set S of h sequences of length w, a string P of length w, and an integer k, such that $1 \leq k \leq h$, find all k-SMEMs between P and S.*

As previously mentioned, the k-MPSC problem does not admit a solution if there exists any positional substring (i, i, P) that is not included in at least k strings from S. On the contrary, the k-SMEM-finding problem always admits a solution–possibly not covering some columns.

Leftmost, Rightmost, and Length-maximal MPSC

Given a panel S and a string P there can exist several distinct k-MPSC of the same size (hence, several solutions to the problem). Since the returned solution might affect the results of downstream applications of k-MPSC, three problems have been identified [10, 11] to constrain (and, possibly, to uniquely identify) the returned solution. As in the original paper, we state the problems in terms of MPSC (hence, 1-MPSC), but they can be generalized to k-MPSC.

For the definition of the problems, given a MPSC C, the i-th positional substring of C, for $1 \leq i \leq |C|$, is the i-th positional substring in the enumeration of the positional substrings of C by increasing the starting positions, while the *length* of C is the sum of the lengths of its positional substrings.

- find a *leftmost MPSC* C of P by S, i.e. a MPSC of P by S such that any i-th substring in C starts at least as early as the i-th substring of every other MPSC of P by S
- find a *rightmost MPSC* C of P by S, i.e. a MPSC of P by S such that any i-th substring in C ends at least as late as the i-th substring of every other MPSC of P by S
- find a *length-maximal MPSC* of P by S, i.e. the MPSC that has the largest length out of all MPSCs of P by S

Given a string P of length w, a set S of h strings of length w, and the PBWT of S, Sanaullah et al. showed that all these problems can be solved in $\mathcal{O}(w)$ time and $\mathcal{O}(n)$ space [11].

2.2 Positional Burrows–Wheeler Transform

The PBWT [4] is a data structure that allows to efficiently perform pattern matching tasks on a set $S = \{S_1, \ldots, S_h\}$ of h binary sequences of length w.

The core data structure is composed of two arrays per each column j: the *prefix array* PA_j and the *divergence array* DA_j. In detail, PA_j stores the permutation of the set $\{1, \ldots, h\}$ induced by the co-lexicographic ordering of prefixes of S up to column $j-1$. More formally, $\mathsf{PA}_j[i]$ is equal to k iff $S_k[1..j-1]$ is the i-th element in co-lexicographical ordered list of prefixes $S_1[1..j-1], \ldots, S_h[1..j-1]$. We note that $\mathsf{PA}_1 = \{1, \ldots, h\}$. $\mathsf{DA}_j[i]$ stores the length of the longest common suffix between the sequence in position i and its predecessor in the co-lexicographic ordering of prefixes up to the $(j-1)$-th column, i.e., $S_{\mathsf{PA}_j[i]}[1..j-1]$ and $S_{\mathsf{PA}_j[i-1]}[1..j-1]$. We note that $\mathsf{DA}_1 = \{0, \ldots, 0\}$. Finally, the PBWT of S is a matrix $\mathsf{PBWT}[1..h][1..w]$ where each column j stores the bits contained in each position j of each input sequence reordered by the permutation induced by PA_j. More formally, if we consider

the input set S as a matrix M and we denote the j-th column of a matrix A by $\mathsf{col}(A)_j$, we have $\mathsf{col}(\mathsf{PBWT})_j[i] = \mathsf{col}(\mathsf{M})_j[\mathsf{PA}_j[i]]$ for all $i = 1..h$ and $j = 1..w$. Durbin [4] prove that we can compute the entire set of PA arrays, the entire set of DA arrays, and the PBWT matrix in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.

2.3 Run-length Encoded PBWT and μ-PBWT

In the seminal paper that first introduced the PBWT, Durbin [4] noted that run-length encoding can be adapted to the PBWT. Later Cozzi et al. [3] and Bonizzoni et al. [1] proposed various data structures to efficiently store and query a run-length encoded PBWT (RLPBWT). In the context of the PBWT, the number of *runs* is equal to the number of length-maximal substrings of equal symbols that appear in the columns of the PBWT. Given r_j as the number of runs in RLPBWT column j, we denote r as $\sum_{1 \leq j \leq w} r_j$.

Here, we consider the RLPBWT implementation of Cozzi et al., which is referred to as μ-PBWT. Similar to the conventional BWT, the SMEMs-finding problem can be solved by computing Matching Statistics for a pattern P against a set of sequences S in a run-length manner.

▶ **Definition 4** (Matching Statistics in the PBWT). *Given a binary panel composed by h sequences $S = \{S_1, \ldots, S_h\}$ of length w and a pattern $P[1..w]$, we define the Matching Statistics of P with respect to S as an array $MS[1..w]$ of* $(\mathsf{row}, \mathsf{len})$ *pairs such that, for each position $1 \leq j \leq w$:*

- $S_{MS[j].\mathsf{row}}[j - MS[j].\mathsf{len} + 1..j] = P[j - MS[j].\mathsf{len} + 1..j]$ *(having a match of length $MS[j].\mathsf{len}$ shared between P and $MS[j].\mathsf{row}$ that ends in j)*
- $P[j - MS[j].\mathsf{len}..j]$ *does not occur as a suffix ending in the j-th column in any sequences of S (left maximality)*
- $MS[j].\mathsf{row} = -$ *and* $MS[j].\mathsf{len} = 0$ *iff* $P[j]$ *does not occur in column j*

Observe that SMEM are computed from the Matching Statistics array. More precisely, a SMEM of length $MS[j].\mathsf{len}$ occurs between P and row $MS[j].\mathsf{row}$, starting from position $j - MS[j].\mathsf{len} + 1$ in P, if $MS[j].\mathsf{len} \neq 0$ and either $j = w$ or $MS[j].\mathsf{len} \geq MS[j+1].\mathsf{len}$. In fact, we cannot extend to the right the considered longest common suffix shared by P and any sequence in S, guaranteeing the right maximality.

As in [3], μ-PBWT computes the Matching Statistics array in $\mathcal{O}(r)$ space by storing only the following data:

- a *mapping structure* to support in constant time the FL (First-to-Last) function used to follow a row in the permutation induced by the PBWT from left to right. Note that μ-PBWT can perform in logarithmic time the reverse of this mapping, following a row from right to left;
- the PA *samples* at run boundary;
- the set of *thresholds*, that identify the positions of the first minimum DA value in the range of each run.

μ-PBWT also stores a small successor data structure, called Φ data structure, that is used to identify the location of the SMEMs in $\mathcal{O}(r)$ space. Using the Φ data structure, we retrieve the previous/next PA/DA value from given a PA/DA value in $\mathcal{O}(\log n/r)$ time [3]. We refer readers to Cozzi et al. paper and to Bonizzoni et al. paper to recall the methods used to compute the Matching Statistics array and the SMEMs.

3 Methods

In this section, we first present a method to extend the Matching Statistics computation to be able to detect k-SMEMs. We then present a novel approach using Matching Statistics to solve some variants of the MPSC problem. The principle underlying our approach is that addressing the k-SMEM problem is essential for dealing with the k-MPSC problem.

3.1 From MS and SMEMs to k-MS and k-SMEMs

In 2023, Tatarnikov et al. [12] extended MONI [9] (an efficient RLBWT [8] and r-index [5, 6] implementation) to demonstrate its capability in computing k-MEMs – which are defined as maximal exact matches of a pattern against a text T that occur at least k time in T.

We recall that FL is the function used to trace from left to right the position of a given row when it is changed by the reordering of rows induced by the PBWT, i.e., computing the position of the bit corresponding to a certain row in a column in the PBWT from the previous one.

▶ **Definition 5** (k-support values). *Given an index column j, a run endpoint index b in the $(j-1)$-th column and the corresponding k-interval $\mathsf{DA}_j[\mathsf{FL}(b)-k+2..\mathsf{FL}(b)+k-1]$, we define $(\mathsf{off}_b, \mathsf{L}_b)$ as its k-support values, where:*

- off_b *stores the offset from* $\mathsf{FL}(b)$ *to get the beginning of the sub-interval of size* $k-1$ *of the k-interval that maximizes the minimum divergence array value d across all the possible sub-intervals of size $k-1$. If this interval starts in* $\mathsf{FL}(b)+1$*, we have* $\mathsf{off}_b = -1$
- $\mathsf{L}_b = d$

In other words, k-support values determine, for a run endpoint b, the sub-interval of size $k-1$ of $\mathsf{DA}_j[\mathsf{FL}(b)-k+2..\mathsf{FL}(b)+k-1]$ which has the longest possible common suffix (of length d) shared by all the $k-1$ rows (plus the previous one by DA array definition) stored in the same sub-interval in PA_j. In addition, due to the definition of DA_j, we can include the row that precedes the interval in this set of rows. In Figure 1 we illustrate an example of k-interval and k-support values.

We can compute the k-support values or at indexing time, accessing the entire PA/DA in constant time, or at querying time using the Φ data structure.

▶ **Theorem 6.** *Given k and $\mathsf{DA}_j[1..h]$ as an array with random access in constant time, we can compute the k-support values in $\mathcal{O}(k)$ time. Instead, if we consider the μ-PBWT Φ functions to access DA_j, these values can be computed in $\mathcal{O}(k \log n/r)$ time.*

The following result shows that adding the computation of the k-support to μ-PBWT does not change the space complexity.

▶ **Lemma 7.** *Given k and the μ-PBWT for a panel of size $n = hw$, we can store all the k-support values in a $\mathcal{O}(r)$ space, having two additional integers at each run boundary.*

Next, to efficiently compute k-SMEMs in a run-length manner, we generalize the definition of Matching Statistics to the k-MS.

▶ **Definition 8** (k-Matching Statistics in the PBWT). *Given a binary panel composed by h sequences $S = \{S_1, \ldots, S_h\}$ of length w, a pattern $P[1..w]$ and a value k, we define the k-Matching Statistics of P with respect to S as an array $k\text{-}MS[1..w]$ of $(\mathsf{row}, \mathsf{len})$ pairs such that, for each position $1 \leq j \leq w$:*

	DA_6	1	2	3	4	5	$col(\mathsf{PBWT})_6$
1	0	0	0	0	1	0	0
2	3	0	1	0	1	0	1
3	5	0	1	0	1	0	1
4	5	0	1	0	1	0	0
5	1	0	1	0	0	0	0
6	4	1	1	0	0	0	1
7	1	1	0	0	1	0	0
8	3	0	1	0	1	0	1

Figure 1 k-support values example with $k = 4$. The figure shows the co-lexicographical ordering up to column 5 used to compute $col(\mathsf{PBWT})_6$ and DA_6. Suppose that some run boundary b in $col(\mathsf{PBWT})_5$ is mapped to $col(\mathsf{PBWT})_6[4]$ (the green 0 in the $col(\mathsf{PBWT})_6$ column). We consider the 4-interval $\mathsf{DA}_6[\mathsf{FL}(b) - k + 2..\mathsf{FL}(b) + k - 1] = \mathsf{DA}_6[2..7]$, circled in green in the DA_6 column. We now consider all the possible subintervals of size $k - 1$, here 3, considering their minimum divergence array value. By definition $\mathsf{DA}_j[i]$ compares row i e row $i - 1$ in the co-lexicographical ordering so, with $k = 3$, we are considering four rows. For example, with $\mathsf{DA}_6[2]$ we are taking into account also row 1 to compute that divergence array value. These subintervals, including the additional rows, are identified by circles in the panel. We are interested in the optimal possible subinterval, so the one that involves the maximum common extension to the left d. In this example, the orchid one, i.e. $\mathsf{DA}_6[2..4]$, is the optimal one, with $d = 3$. In conclusion, we store the offset $\mathsf{off}_b = 2$ and $\mathsf{L}_b = 3$.

- $S_{k\text{-}MS[j].\mathsf{row}}[j - k\text{-}MS[j].\mathsf{len} + 1..j] = P[j - k\text{-}MS[j].\mathsf{len} + 1..j]$ *(there exists a match of length* $k\text{-}MS[j].\mathsf{len}$ *shared between* P, $k\text{-}MS[j].\mathsf{row}$ *and at least other* $k - 1$ *rows in* S *that ends in* j*)*
- $P[j - k\text{-}MS[j].\mathsf{len}..j]$ *does not occur as a suffix ending in the* j*-th column in any subset* S' *of sequences of* S *with* $|S'|$ *greater or equal of* k *(left maximality of the match)*
- $k\text{-}MS[j].\mathsf{row} = -$ *and* $k\text{-}MS[j].\mathsf{len} = 0$ *iff* $P[j]$ *does not occur at least* k *time in column* j

Given μ-PBWT, as described in Section 2.3, we can extend it with the k-support values and solve the k-SMEMs problem in a run-length encoding manner. Note that the k-support values can be pre-computed and queried in constant time, or they can be retrieved each time according to the complexity in Theorem 6. Observe that the computation of $k\text{-}MS.\mathsf{len}$ array involves updating column j, beginning with $j = 1$, and necessitates the modification of new additional arrays: len_k and len_t. We add a *support index* s_j, which is a position in the column j to verify whether at least k rows have the same left-maximal match up to the j-th column. Then:

- len_k stores in position j the length of the left-maximal matches shared by at least k rows (defined by the sub-interval that we get from s_j) in the panel
- len_t acts as the classical $MS.\mathsf{len}$ array. In column j, $\mathsf{len}_t[j]$ stores the length of the semi-left maximal match shared between $k\text{-}MS[j].\mathsf{row}$ and a row in the input panel S. We say *semi-left maximal* because, unlike classical the MS array, we cannot extend to the left a match if a column i, such that $1 \leq i < j$, does not contain at least k symbols $P[i]$,

We do not need to store these arrays in (entirely) memory – rather we store two variables with their values in column j. See Figure 2 for an example of len_k and len_t.

3.1.1 Computing k-MS and k-SMEMs

The computation of the k-MS array involves iterating starting from position $j = 1$ within both the array and the pattern P. Initially, if the first column contains at least k occurrences of the symbol $P[1]$, then we assign the values $k\text{-}MS[1].\mathsf{row} = s$ and $k\text{-}MS[1].\mathsf{len} = 1$, where

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	1	0	0	0	0	0	0	0	1	1	0	1
2	1	0	0	1	1	0	0	1	0	0	0	0	0	1	1
3	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
4	1	0	0	1	1	0	0	1	0	0	0	1	0	0	1
5	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
6	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
7	0	1	0	1	0	1	0	0	0	0	0	1	0	0	1
8	0	1	0	1	0	1	0	0	0	0	0	0	1	0	1
9	0	1	0	0	1	0	0	0	0	1	1	1	0	0	1
10	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
11	0	1	0	1	0	0	0	0	1	0	0	0	0	1	1
12	0	1	0	0	1	0	0	0	0	0	1	1	0	0	0
13	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
14	0	1	0	0	1	0	0	0	1	0	1	1	0	0	1
15	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
16	0	1	0	0	0	0	0	0	1	0	0	0	1	0	1
17	0	1	0	1	0	0	0	0	0	0	0	1	1	0	1
18	1	1	0	0	0	1	0	0	0	0	0	1	1	0	1
19	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
20	0	1	1	0	1	0	1	0	0	0	0	0	1	0	1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	0	1	0	0	1	0	1	0	0	0	1	1	1	0	1
row	17	17	17	16	14	14	–	19	12	12	12	12	18	18	18
len_t	1	2	3	4	5	6	0	1	2	3	4	5	2	3	4
len_3	1	2	3	4	5	6	0	5	6	5	2	3	7	8	9
len	1	2	3	4	5	6	0	1	2	3	2	3	2	3	4

Figure 2 Example of 3-SMEMs results. We consider an input matrix M that represents a set of 20 sequences of length 15 and a pattern P of the same length. We show the 3-SMEMs using the same colour in M and P. In addition, we show the k-MS array (with underlined all the values that represent a SMEM), the len_3 array, and the len_t array.

s belongs to the set S with the condition that $S_s[1] = P[1]$. If this condition is not met – that is, if there are fewer than k instances of $P[1]$ in the first column – we set k-$MS[1].\mathsf{row}$ to a placeholder value (indicated by $-$) and k-$MS[1].\mathsf{len}$ to 0.

We now describe how to update k-$MS[j]$ from k-$MS[j-1]$ for $j > 1$. To update the column of k-MS for j we need to follow k-$MS[j-1].\mathsf{row}$ in the permutation induced by PA_j to extend to column j the left-maximal match obtained in column $j-1$ if it is possible. Given that k-$MS[j-1].\mathsf{row}$ is mapped in row i of the column j (via the FL function) in the PBWT, there are two cases that we have to consider: (1) we have a mismatch between the pattern P and row i in the j-th column, and (2) we have a match between the pattern P and row i in the j-th column. Within these two cases, there are some additional sub-cases that we have to consider. In what follows, we describe how to address these cases, and then, we show how to use len_k and len_t to compute k-MS and k-SMEMs.

Case 1: A mismatch occurs

We assume that we have a mismatch in the j-th column, i.e., $\mathsf{col}(\mathsf{PBWT})_j[i] \neq P[j]$. If $\mathsf{col}(\mathsf{PBWT})_j$ does not contain at least k occurrences of the symbol $P[j]$ then we have a *complete mismatch*. We represent this with k-$MS[j].\mathsf{row} = -$ and $\mathsf{len}_t[j] = \mathsf{len}_k[j] = 0$, resetting the

k-MS computation in the next column – implying that we restart the computation from column $j+1$ as we were in column 1. Otherwise, if $k\text{-}MS[j].\mathsf{row}$ (where $k\text{-}MS[j].\mathsf{row}$ and $MS[j].\mathsf{row}$ are equal) and $\mathsf{len}_t[j]$ (which corresponds to the traditional $MS[j].\mathsf{len}$ value), then we apply the approach of Cozzi et al. [3]. This update involves the selection of a new run boundary b. We use the k-support values to update $\mathsf{len}_k[j]$ and s_{j+1} as follows: $\mathsf{len}_k[j] = L_b$ and $s_{j+1} = \mathsf{FL}(b) - \mathsf{off}_b$. In this way, we consider the sub-interval of size k, consisting of rows that share the longest common suffix of length L_b up to column j.

Case 2: A match occurs

Next, we assume that we have a match in the j-th column, i.e., $\mathsf{col}(\mathsf{PBWT})_j[i] = P[j]$. If $\mathsf{col}(\mathsf{PBWT})_j$ does not contain at least k occurrences of the symbol $P[j]$ we have an *unfeasible match*. We handle this case as in the *complete mismatch* case described above. We recall that we use the support index s_j to identify the starting position of the sub-interval of size k in column j of the PBWT in which the rows share the longest common suffix up to column j. Observe that if we have $\mathsf{col}(\mathsf{PBWT})_j[s_j] = \cdots = \mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1] = P[j]$ then we have a left-maximal match shared by at least k rows. It follows that we can update the Matching Statistics in the same manner as in the μ-PBWT: $k\text{-}MS[j].\mathsf{row} = k\text{-}MS[j-1].\mathsf{row}$, $s_{j+1} = \mathsf{FL}(s_j)$, $\mathsf{len}_t[j] = \mathsf{len}_t[j-1] + 1$, and $\mathsf{len}_k[j] = \mathsf{len}_k[j-1] + 1$. Informally, we follow the same sub-interval (by FL function) in column $j+1$, updating all the length values by 1 (due to the match) to consider the next column. We note that the condition $\mathsf{col}(\mathsf{PBWT})_j[s_j] = \cdots = \mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1] = P[j]$ can be checked without scanning entirely $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j + k - 1]$. In fact, the condition is satisfied iff $\mathsf{col}(\mathsf{PBWT})_j[s_j]$ and $\mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1]$ lay in the same run and we can test this fact in logarithmic time (in the number of runs of the column j-th).

Finally, it is necessary to address the sub-cases that arise when there are symbols within $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j + k - 1]$ that do not match $P[j]$. Again, this fact can be checked by looking at the runs of the first and the last symbols in this interval. If $\mathsf{col}(\mathsf{PBWT})_j[s_j]$ and $\mathsf{col}(\mathsf{PBWT})_j[s_j + k - 1]$ lay on different runs it means that we have at least a run of symbols that differ from $P[j]$ between these two runs. Thus, we need to use a different support index. To identify the new support index s_{j+1} it is necessary to trace a new row. Recall that all the information used to update the k-MS are stored at a run boundary, and therefore, we need to select a new run boundary b such that $\mathsf{col}_j(\mathsf{PBWT})[b] = P[j]$. This b is selected as in the mismatch case, assuming to consider i as the index of the first mismatch in $\mathsf{col}(\mathsf{PBWT})_j[s_j..s_j + k - 1]$. At this point we can update $\mathsf{len}_t[j]$. For this purpose, we compute the length of the common suffix up to the j-th column between $k\text{-}MS[j-1].\mathsf{row}$ (that we are currently following due to the match) and $\mathsf{PA}_j[b]$ and compare it to $\mathsf{len}_t[j-1]+1$. We select the minimum of these two lengths to retain only the suffix that encompasses k rows. So, if we denote $\mathsf{lcs}_j(A, B)$ as the longest common suffix up to the j-th column shared by rows A and B, then we have $\mathsf{len}_t[j] = \min(\mathsf{lcs}_j(k\text{-}MS[j-1].\mathsf{row}, \mathsf{PA}_j[b]), \mathsf{len}_t[j-1] + 1)$. Then we update $k\text{-}MS[j].\mathsf{row}$ using the prefix array samples as in μ-PBWT. Moreover, to update s_{j+1} and $\mathsf{len}_k[j]$ we use the k-support values as follows: $\mathsf{len}_k[j] = L_b$ and $s_{j+1} = \mathsf{FL}(b) - \mathsf{off}_b$. In every case mentioned that requires jumping to a new row, we note that if there is no run available for the jump (for instance, when there are only two runs), then the algorithm followed is akin to what is done in the case of a complete mismatch.

Filling the k-MS array and computing k-SMEMs

To account for the lengths of all matches, we formulate the $k\text{-}MS.\mathsf{len}$ as follows: $k\text{-}MS[j].\mathsf{len} = \min(\mathsf{len}_t[j], \mathsf{len}_k[j])$, for all $j = 1..w$. This step can be performed after processing each column j. Finally, similar to the computation of SMEMs [1, 3], a k-SMEM of length $k\text{-}MS[j].\mathsf{len}$ occurs between P and row $k\text{-}MS[j].\mathsf{row}$, starting from position $j - k\text{-}MS[j].\mathsf{len} + 1$ in P, if $k\text{-}MS[j].\mathsf{len} \neq 0$ and either $j = w$ or $k\text{-}MS[j].\mathsf{len} \geq k\text{-}MS[j+1].\mathsf{len}$.

▶ **Theorem 9.** *Given a set S of h sequences of length w, an integer k, and query z of size w, we can compute the k-SMEMs for z using $\mathcal{O}(r)$ space.*

Providing a similar bound for the time complexity of computing k-SMEM in the μ-PBWT is an open problem that warrants consideration. We conjecture that it can be done in $\mathcal{O}(w \log r)$ time. Lastly, we illustrate the results of computing 3-SMEMs in Figure 2.

3.1.2 From k-MS to k-MPSC

We show now that we can build a solution for k-MPSC by using the k-MS array. We recall that each positional substring that belongs to a k-MPSC is contained in a k-SMEM. By the non-inclusion property of k-SMEM, each starting position of a k-SMEM is covered by one and only one k-SMEM. Combining the above two properties, it follows that there exists a k-MPSC in which each substring is a prefix of a k-SMEM. In addition, there exists only one k-SMEM that covers $P[w]$. Therefore, we can start building a solution to k-MPSC by adding this k-SMEM as a positional substring. We can now proceed iterating the process of including each time the single left-maximal match (covered by at least k rows in S) that ends in the column j prior to the starting column $j+1$ of the last positional substring added to k-MPSC. This left-maximal match is represented by k-MS values in position j, and it is unique according to the definition of k-MS. The minimality condition is guaranteed by the fact that we cannot have a better solution, i.e., a single left-maximal match γ that covers two of the left-maximal matches, α and β, that were already added to k-MPSC. To see this, consider that if such a γ exists, it must begin prior to α owing to the left-maximal property and extend at least up to the identical ending position β at the j-th index. In this case, the definition of k-MS property implies that γ should be identified in position j of the k-MS array. This is a contradiction because in position j of k-MS array we already selected the left-maximal match β. Notice that this procedure extends Algorithm 1 to k-MPSC, as demonstrated in the following section where we prove its efficacy in computing the leftmost MPSC.

3.2 Solving the Leftmost, Rightmost, and Length-maximal MPSC Problems with MS and SMEMs

We now show how to use MS and SMEMs to solve the leftmost, rightmost, and length maximal MPSC problems. We first show that Algorithms 1 and 2 solve the leftmost (Lemma 10) and rightmost (Lemma 11) MPSC problems, respectively, in sublinear space. We recall that r denotes the number of runs in the PBWT of a set S of h sequences of length w, which is upper bounded by their size $n = h \cdot w$. The proofs of Lemma 10 and Lemma 11 are based on the non-inclusion property of SMEMs, which implies that the starting positions and ending positions of the set of SMEMs are unique. In other words, there cannot exist two SMEMs that start in the same position in the pattern P and two SMEMs that end in the same position in the pattern P.

Algorithm 1 Leftmost MPSC by MS.

```
1: function LeftMost(MS)
2:     j ← w                                  ▷ |MS| = w
3:     j' ← 0
4:     while j ≠ 0 do
5:         j' ← j − MS[j].len
6:         report (j' + 1, j, MS[j].row)
7:         j ← j'
```

Algorithm 2 Rightmost MPSC by MS.

```
1: function RightMost(MS)
2:     i ← 1                                  ▷ |MS| = w
3:     for j = 1 → w − 1 do
4:         if MS[j].len ≥ MS[j+1].len then
5:             report (i, j, MS[j].row)
6:             i ← j + 1
7:     report (i, w, MS[w].row)
```

▶ **Lemma 10** (Leftmost MPSC). *Given a panel S and the MS array of a pattern P with respect to S, Algorithm 1 computes the leftmost MPSC C of P by S in time $\mathcal{O}(|C|)$ and $\mathcal{O}(r)$ space.*

Proof. We seek a MPSC such that each i-th positional substrings starts not after any other i-th positional substring in a MPSC. It is easy to see that a leftmost MPSC is composed of positional substrings that are prefixes of SMEMs – or, in other words, that are left-maximal. Otherwise, we can extend the positional substring to the left, contradicting the assumption of having a leftmost MPSC. Scanning the MS array from right to left, we consider the SMEM that ends in the last position of the pattern. This SMEM is in the set of the leftmost MPSC by definition, and the fact that it is the only one that includes position w. We can compute its starting position j as $w - MS[w].\mathsf{len} + 1$ and we can add $(j, w, MS[w].\mathsf{row})$ to the set C. The set C is a leftmost MPSC of the columns j to w. A leftmost MPSC of columns 1 to $j-1$ must include the left-maximal match that ends at $j-1$. Given that $j' = (j-1) - MS[j-1].\mathsf{len} + 1$, by definition of SMEM, it follows that we cannot have a SMEM that includes $MS[j-1].\mathsf{row}$ in position $j-1$ and starts before j'. Thus j' is the starting position of the next positional substring (of length $MS[j-1].\mathsf{len}$) in the leftmost MPSC C of the columns j' to w. Iterating the previous procedure until reaching column 1 gives the leftmost MPSC. Since each iteration adds a positional substring to the cover with a single constant-time access of the MS array, the algorithm runs in time proportional to $|C|$. ◀

▶ **Lemma 11** (Rightmost MPSC). *Given a panel S of w-length strings and the MS array of a pattern P with respect to S, Algorithm 2 computes the rightmost MPSC of P by S in time $\mathcal{O}(w)$ and $\mathcal{O}(r)$ space.*

Proof. We are interested in a MPSC where each i-th positional substring ends no earlier than any other i-th positional substring within it. Symmetrically to the leftmost MPSC, the positional substrings of a rightmost MPSC are suffixes of SMEMs. Given the MS array, a position j is the ending position of a SMEM iff $MS[j].\mathsf{len} \geq MS[j+1].\mathsf{len}$ or $j = w$. For simplicity, in this proof we assume that $MS[w+1].len = 0$. The construction of the rightmost MPSC is symmetric to that of the leftmost MPSC. As in the leftmost MPSC, we are not interested in overlaps between positional substrings. To compute the rightmost MPSC, we scan from left to right the MS array. By definition, the SMEM that starts in the first column is in the set of rightmost MPSC, being the only one that includes the first position. Denoting j as the ending position of this SMEM, we add the positional substring $(1, j, MS[j].\mathsf{row})$ to the set C of the rightmost MPSC. At each position j where $MS[j].\mathsf{len} \geq MS[j+1].\mathsf{len}$, there are no other right-maximal matches, meaning there cannot exist an SMEM containing $MS[j].\mathsf{row}$ at position j that extends beyond position j. Thus, js are the ending position

M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0
2	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0
3	1	0	1	0	1	0	0	1	0	0	0	1	1	0	0
4	1	1	1	0	1	1	1	1	0	0	1	0	0	0	0
5	0	1	0	1	0	0	0	0	1	1	0	1	0	0	1
6	1	0	1	0	1	1	1	1	0	0	1	1	0	0	0

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
row	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4

(a) Splitting of a SMEM into substrings from leftmost/rightmost MPSC. We show a panel M and a pattern P with the corresponding MS array. In the panel, we have dashed circles for the leftmost MPSC and dotted circles for the rightmost MPSC. With the same colour, we identify a SMEM.

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
row	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4
j/j'	←	2	←	←	←	6	←	←	←	←	11	←	←	←	15

(b) Example of computing leftmost MPSC (identified by circles in the P row) by the MS array. In the last line we show the jumps used to skip the overlaps as in Algorithm 1.

MS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P	1	1	1	0	1	0	1	1	0	0	1	1	1	0	0
row	6	4	4	4	4	3	1	1	1	1	6	6	3	3	3
len	1	2	3	4	5	4	5	6	7	8	5	6	2	3	4
i	1	–	–	–	–	6	–	–	–	–	11	–	13	–	–

(c) Example of computing rightmost MPSC (identified by circles in the P row) by the MS array. For the sake of simplicity, in the last line, we show the updating of value i as in Algorithm 2.

Figure 3 Example of the relationship between leftmost/rightmost MPSC and SMEMs on the same set of sequences of [11].

of any other positional substring $(i+1, j, MS[j].\mathsf{row})$ in the set C, where i is the ending column of the last substring added to C. Since each position of the MS array is scanned once, and as each iteration requires only a single constant-time access of the MS array, the algorithm runs in time proportional to w. ◀

Figure 3 shows how our algorithms compute the leftmost and the rightmost MPSC.

We now consider the problem of finding a length-maximal MPSC. Sanaullah et al. [11] showed the length-maximal MPSC problem can be solved in $\mathcal{O}(n + |Q|)$ space given that the leftmost MPSC, the rightmost MPSC, and the set Q of SMEMs have been computed in $\mathcal{O}(n)$ space, while the algorithm used to combine all of them to find a length-maximal MPSC requires $\mathcal{O}(|Q|)$ space. We showed that we can compute the leftmost and the rightmost MPSCs in $\mathcal{O}(r)$ space (Lemma 10 and Lemma 11) from the MS array. We also showed in Section 2.3 that the MS array and the set of SMEMs can be computed in $\mathcal{O}(r)$ space using the μ-PBWT. As a consequence, the following lemma holds.

▶ **Lemma 12.** *Given μ-PBWT for a set S of h strings of length w, a string P of length w, and the set Q of SMEMs shared by S and P, a length-maximal MPSC of P by S can be computed in $\mathcal{O}(r + |Q|)$ space.*

4 Results

We implemented our methods for computing k-MS and k-SMEMs with and without pre-computed k-support values, and compared these to the k-MPSC implementation by Sanaullah et al. [11]. Hereon, we refer to this method as `k-MPSC`.

4.1 Datasets

We evaluated the execution time and maximum memory usage for indexing and querying of all these methods using biallelic chromosome panels from the 1000 Genomes Project (1KGP) [13]. All the data is publicly available at `https://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/`. We selected the panels for chromosomes 22, 18, and 2. These panels have 5,008 samples/rows and between 1M and 6M variations/columns. From these panels, we extracted 30 rows to use them as queries, hence we consider input panels with 4978 rows. The performance metrics were computed using `/usr/bin/time` on a machine equipped with an Intel Xeon CPU E5-4610 v2 (2.30GHz), 256GB RAM and 8GB of swap, running Ubuntu 20.04.6 LTS.

4.2 Implementation Details and Experimental Setup

We augmented μ-PBWT implementation with the algorithms to compute k-SMEMs. We made all source code publicly avaliable at `https://github.com/dlcgold/muPBWT/tree/k-smem` (with pre-computed k-support values) and `https://github.com/dlcgold/muPBWT/tree/k-smem-live` (without pre-computed k-support values).

`k-MPSC` is written in `C++14` and necessitated a few modifications to the code. It was not possible to index panels as large as the ones from the 1KGP due to design choices on the dynamics allocation of the data structure. Moreover, `k-MPSC` implementation halted if an unfeasible column was encountered (a column with less than k symbols), which frequently occurred only after very few columns. To make a meaningful comparison, we edited the source code to solve the allocation issue and to restart the computation after an unfeasible column is encountered.

Empirical experimental performances for the computation of a leftmost MPSC, a rightmost MPSC, and a length-maximal MPSC (from the MS array and the set of SMEM) are directly proportional to the experimental results in Cozzi et al. [3]. Therefore, no additional dedicated experiments were conducted.

4.3 Results on 1000 Genomes Project Data

Table 1 reports the results of the indexing task and of the querying task with 30 queries. As anticipated, pre-computing the k-support values leads to an increase in computation times by up to approximately 100% compared to the variant without pre-computed values. We note that, due to the average number of runs in each column, we only need to store short bit-compressed integer vectors for the k-support values, implying that only a $\sim$5% increase in memory usage is needed for the implementation without these values stored. A fairly obvious note that warrants comment is that for $k = 1$, we do not require k-support values, thus the memory usage remains the same both with and without them. Regarding `k-MPSC`, the indexing phase consists of the PBWT computation of both panel and queries. Since `k-MPSC` needs a full PBWT, it requires almost two orders of magnitude more memory than both of our approaches. This is unsurprising in light of the work of Durbin [4], which stated that the whole set of data structures for PBWT require $13n$ bytes to be queried.

Table 1 Indexing and querying wall clock time and max memory usage comparison on chromosomes 2/18/22 panels from 1KGP, with 4978 rows and 30 queries.

			Wall Clock Time (seconds)			Max Memory usage (GB)		
			μ-PBWT			μ-PBWT		
Chr.	Task	k	(pre)	(no-pre)	`k-MPSC`	(pre)	(no-pre)	`k-MPSC`
2	*Index*	1	1381	1384	-	6.45	6.45	-
		50	1643	1387	-	6.62	6.45	-
		200	2798	1418	-	6.62	6.45	-
	Querying	1	254	247	-	5.87	5.87	-
		50	1687	2471	-	6.57	6.42	-
		200	3563	11834	-	8.45	8.31	-
18	*Index*	1	425	424	5750	3.26	3.26	168.58
		50	697	450	5750	3.36	3.26	168.58
		200	807	429	5750	3.36	3.26	168.58
	Querying	1	70	70	128	1.92	1.92	171.19
		50	739	817	229	2.13	2.08	171.30
		200	1740	4263	119	2.74	2.71	171.22
22	*Index*	1	191	189	2616	1.77	1.77	79.01
		50	304	193	2616	1.84	1.77	79.01
		200	400	194	2616	1.84	1.77	79.01
	Querying	1	31	32	130	0.98	0.98	82.31
		50	336	392	171	1.11	1.07	82.85
		200	726	2054	133	1.47	1.44	82.48

Regarding the querying performance results, the computation of k-support values during query time leads to an increase in execution time, which scales with the value of k due to the use of the Φ data structure. When k was equal to 200, the k-SMEMs computation requires up to a third of the time with the pre-computed values. Regarding `k-MPSC`, with random access in constant time to the complete set of data structures of the PBWT, we observe that it runs up to 20 times faster than μ-PBWT with pre-computed values and up to 60 times faster than the variant with query-time k-support values computation. Moreover, the experiments confirm that the k-MPSC algorithm does not linearly scale on k, as explained in Section 2.1. Regarding memory usage for querying, a similar analysis apply as for the indexing phase with the additional factor that we also have in memory the queries for the μ-PBWT. In addition, we suspect that the slight increase in memory usage with larger k values is due to the support variables used. Since in instances where the computation of MS requires pointer adjustments because there are not k equivalent values in a designated interval, we have to compute $\mathsf{lcs}(A, B)$.

Due to the memory usage of `k-MPSC`, we were unable to evaluate the performance on the larger panels, such as the one related to the chromosome 2 in the 1000 Genomes Project data. To ensure a comprehensive overview, Table 1 includes the outcomes achieved by μ-PBWT on the largest dataset (chromosome 2), which `k-MPSC` could not process within the allocated memory constraints. This demonstrates the advantages of using μ-PBWT which requires a sublinear amount of memory.

5 Conclusions

In this paper, we address the theoretical aspects of the haplotype threading problem, primarily aiming to offer practical solutions capable of scaling to the large biological datasets currently stored in biobanks. In particular, we presented two distinct results. First, we adapted the run-length encoding paradigm for the PBWT to compute k-Matching Statistics (k-MS) and the k-SMEMs in sublinear space. Next, we show that computing k-MS provides a theoretical framework to solving the k-Minimal Positional Substring Cover problem as well as all the variants of the MPSC problem in sublinear space. Our experimental results decisively show that our method achieves a significant reduction in memory usage for computing the k-MPSC problem, reducing it by no less than two orders of magnitude relative to the approach by Sanaullah et al. [11]. This enables our approach to scale to large datasets collected in contemporary biobanks. In turn, the development of μ-PBWT opens up unprecedented opportunities for comprehensive genetic studies – e.g., for genotyping and imputation workflows [14] – and exploration on a large scale.

References

1 Paola Bonizzoni, Christina Boucher, Davide Cozzi, Travis Gagie, Dominik Köppl, and Massimiliano Rossi. Data Structures for SMEM-Finding in the PBWT. In *International Symposium on String Processing and Information Retrieval*, pages 89–101. Springer, 2023.

2 Davide Cozzi. muPBWT k-SMEM. Software, European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement PANGAIA No. 87253, swhId: `swh:1:dir:d3467768a54423c8294abfc44f87f18705b3ed02`, (visited on 04/06/2024). URL: `https://github.com/dlcgold/muPBWT/tree/k-smem`.

3 Davide Cozzi, Massimiliano Rossi, Simone Rubinacci, Travis Gagie, Dominik Köppl, Christina Boucher, and Paola Bonizzoni. μ-PBWT: a lightweight r-indexing of the PBWT for storing and querying UK Biobank data. *Bioinformatics*, 39(9):btad552, 2023.

4 Richard Durbin. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics*, 30(9):1266–1272, 2014.

5 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1459–1477. SIAM, 2018.

6 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *Journal of the ACM*, 67(1):2:1–2:54, 2020. `doi:10.1145/3375890`.

7 Na Li and Matthew Stephens. Modeling linkage disequilibrium and identifying recombination hotspots using single-nucleotide polymorphism data. *Genetics*, 165(4):2213–2233, 2003.

8 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Combinatorial Pattern Matching: 16th Annual Symposium, CPM 2005, Jeju Island, Korea, June 19-22, 2005. Proceedings 16*, pages 45–56. Springer, 2005.

9 Massimiliano Rossi, Marco Oliva, Ben Langmead, Travis Gagie, and Christina Boucher. MONI: A Pangenomic Index for Finding Maximal Exact Matches. *Journal of Computational Biology*, 29(2):169–187, 2022.

10 Ahsan Sanaullah, Degui Zhi, and Shaoije Zhang. Haplotype threading using the positional Burrows-Wheeler transform. In *22nd International Workshop on Algorithms in Bioinformatics (WABI 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

11 Ahsan Sanaullah, Degui Zhi, and Shaojie Zhang. Minimal positional substring cover is a haplotype threading alternative to Li and Stephens Model. *Genome Research*, 33(7):1007–1014, 2023. `doi:10.1101/gr.277673.123`.

12 Igor Tatarnikov, Ardavan Shahrabi Farahani, Sana Kashgouli, and Travis Gagie. MONI Can Find k-MEMs. In *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

13 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526:68–74, 2015.

14 Naga Sai Kavya Vaddadi, Taher Mun, and Ben Langmead. Minimizing reference bias with an impute-first approach. *bioRxiv*, 2023. doi:10.1101/2023.11.30.568362.