mcPBWT: Space-efficient Multi-column PBWT Scanning Algorithm for Composite Haplotype Matching

Pramesh Shakya[1] Ardalan Naseri[2] Degui Zhi[2,*]
Shaojie Zhang[1,*]

[1]Department of Computer Science
University of Central Florida, Orlando FL 32816, USA
[2]School of Biomedical Informatics
University of Texas Health Science Center at Houston, Houston, TX 77030, USA

Abstract

Positional Burrows-Wheeler Transform (PBWT) is a data structure that supports efficient algorithms for finding matching segments in a panel of haplotypes. It is of interest to study the composite patterns of multiple matching segments or blocks arranged contiguously along a same haplotype as they can indicate recombination crossover events, gene-conversion tracts, or, sometimes, errors of phasing algorithms. However, current PBWT algorithms do not support search of such composite patterns efficiently. Here, we present our algorithm, mcPBWT (multi-column PBWT), that uses multiple synchronized runs of PBWT at different variant sites providing a "look-ahead" information of matches at those variant sites. Such "look-ahead" information allows us to analyze multiple contiguous matching pairs in a single pass. We present two specific cases of mcPBWT, namely *double-PBWT* and *triple-PBWT* which utilize two and three columns of PBWT respectively. *double-PBWT* finds two matching pairs' combinations representative of crossover event or phasing error while *triple-PBWT* finds three matching pairs' combinations representative of gene-conversion tract.

*Corresponding authors. Email: degui.zhi@uth.tmc.edu; shzhang@cs.ucf.edu.

1 Introduction

Durbin's PBWT (positional Burrows-Wheeler Transform) [1] is an efficient data structure that operates on a panel of haplotypes that are bi-allelic to find all the matching segments of a given minimum length threshold, L. In addition to that, it's also capable of finding matches for a query haplotype against a reference haplotype panel and other useful compression algorithms.

Given the linear time complexity of PBWT algorithms, they scale well to large datasets. Because of this, they have widely been incorporated in state-of-the-art statistical phasing and imputation tools [2, 3, 4]. PBWT algorithms have also been used to find identical-by-descent (IBD) [5, 6, 7] segments shared among individuals of a population. IBD segments are segments of chromosome shared among individuals such that they share a most recent common ancestor. Numerous features of IBD segments including their counts, length distribution, etc have been studied as they reveal useful information of the population history, selection pressure, and the disease loci [8]. Many other variations of the PBWT algorithm have been developed that tackle variety of problems. gPBWT [9] provides a method to efficiently query graph-encoded haplotypes, d-pbwt [10] provides efficient algorithms for query haplotype insertion and deletion, mPBWT [11] provides algorithms to deal with multi-allelic panels, [12] allows wildcard characters in the PBWT panel to study relative fitness of genomic variants, and cPBWT [13] and related works [14, 15] extend pairwise segment matching to multi-way matching, i.e., clusters of haplotype matches.

It is of interest to study composite haplotype matching patterns. For example, two long segment matches of a haplotype that are adjacent to each other may indicate a recombination event or an error of the phasing method. Another example is a combination of three segment matches, two long ones between the same pair of haplotypes, surrounding a relatively short one in the middle, that is a hallmark of a gene conversion. However, most existing problem formulations of PBWT algorithms are to find single matching segments between pairs of haplotypes or a single matching block among a cluster of haplotypes.

Recently, bi-directional PBWT [16] (or bi-PBWT) was the first to study composite matching patterns of more than one matching blocks. bi-PBWT finds all matches between sufficiently long matching blocks at both sides of a site, with a small gap of tolerance. The bi-PBWT algorithm is a two-pass scanning algorithm, first scanning backwards, storing the reverse PBWT data structure, and then a second pass scanning forward and makes the block matching. We wish to generalize the two-pattern matching problem to more, and with a more general definition of the connections between individual matching patterns.

Here, we formulate the problem of composite haplotype matching. Conceptually, the goal of composite haplotype matching problem is to find a number of pairwise matching segments or matching clusters, each one is long enough, with small enough gaps/overlaps, and the haplotype IDs of these segments satisfy certain condition (e.g., having a single id shared with all segments, and the other IDs may or may not belong to the same individual). The phasing error pattern, recombination, and the gene conversion each can be seen as special cases of such composite haplotype matching patterns.

In this paper, we introduce a space-efficient algorithm, mcPBWT, that utilizes two or more synchronized scans over multiple columns of PBWT to compare and analyze multiple sets of matches. While a naive solution in the style of bi-PBWT that stores pre-computed PBWT is time-efficient, the space-efficiency of pre-computed panel can be inconveniently large for biobank-scale data. Our algorithm's multi-column idea allows various columns of PBWT to exchange information and integrate multiple single matches without a large memory-footprint or disk-usage. The algorithm also makes a single pass of a haplotype panel. This online nature and generalizability of the algorithm will provide an efficient way of studying complex set of matches.

2 Preliminaries

2.1 PBWT Overview

PBWT (positional Burrows-Wheeler Transform) is an efficient data structure that finds all matches of user-specified minimum length (L) in an efficient manner given a panel of haplotype sequences. In his paper, Durbin [1] defines a *haplotype panel* X as a set of M haplotype sequences $x_i \in X$, $i = 0, 1, 2, ..., M-1$. Each sequence x_i has N SNP sites indexed by k, $k \in \{0, 1, ...N-1\}$. All the sites are assumed to be bi-allelic, namely $x_i[k] \in \{0, 1\}$. *Locally maximal match* is defined as a match between two haplotype sequences s and t from k_1 to k_2 such that, $s[k_1 - 1] \neq t[k_1 - 1]$, or $k_1 = 0$ and $s[k_2] \neq t[k_2]$, or $k_2 = N$. A match is a *long match* if it is locally maximal and its length satisfies a user-specified length threshold. *Prefix array* a contains $N+1$ reverse prefix sorted orderings of the sequences, one for each $k \in 0...N$. It can also be thought of as a permutation of indices of the haplotype sequences that range from 0 to $M-1$ for every $k \in \{0, 1, 2...N\}$. a_k is the k-th reverse sorted ordering of the haplotype sequences up to the site $k-1$. In any a_k, adjacent sequences are maximally matching until k. y_i^k is the i-th sequence in $a_k, y_i^k = x_{a_k}[i]$. The divergence array d_k stores the starting position of locally maximal matches ending at k between a sequence and the preceding sequence in a_k.

2.2 Composite Haplotype Matching

Here, we generalize the notion of haplotype matching in a panel. A *single haplotype match pattern* (or single match) in a haplotype panel is defined as $p = (c, k_1, k_2)$, where c is a subset of the total set of haplotype indexes $C = \{0, ..., M-1\}$, and the haplotype sequences match between sites k_1 and k_2: $x_i[k_1, k_2) = x_j[k_1, k_2)$, for any $i, j \in c$. Here, the *length* of the pattern is $L(p) = |k_2 - k_1|$, and the *width* of the pattern is $W(p) = |c|$. We can also denote the *sequence id set* of p as $c(p) = c$, the *left boundary* of p as $l(p) = k_1$, and the *right boundary* as $r(p) = k_2$. In general, $|c| \geq 2$ indicates a cluster of haplotypes matching. For pairwise matching, $|c| = 2$. The problem of single pattern haplotype matching is, given a predefined length cutoff L and width cutoff W, find all patterns p, such that $L(p) \geq L$ and $W(p) \geq W$ in a haplotype panel.

Further, for two single haplotype match patterns p and q, we define q *g-follows* p if they are adjacent, i.e., the *gap* (or *overlap)* between them, $g(p, q) = l(q) - r(p)$, is small: $|g(p, q)| \leq g$ and some haplotypes are shared among their sequence id sets $c(p) \cap c(q) \neq \varnothing$.

With that, we define a *composite haplotype match pattern* in a haplotype panel as a series of B single matches, $\mathcal{P} = \{p_b\}, b = 0...B-1$, that g-follow each other, i.e., p_i g-follows p_{i-1}, for $i = 1..B-1$, and they share some common haplotypes $c(\mathcal{P}) = \bigcap_{b=0}^{B-1} c(p_b) \neq \varnothing$. We call B the *span* of $\mathcal{P}$, and $c(\mathcal{P})$ the *thread* of $\mathcal{P}$.

The problem of composite haplotype matching pattern is, given a predefined set of length cutoffs $\{L_b\}, b = 0...B-1$, width cutoff W, gap tolerance g, the span B, the thread width w, find all composite patterns $\mathcal{P} = \{p_b\}, b = 0..B-1$ such that $L(p_b) \geq L_b$, $W(p_b) \geq W$, p_i g-follows p_{i-1}, for $i = 1..B-1$, and $|c(\mathcal{P})| = w$. Of course, it is possible to specify different width cutoffs and gap tolerances for individual single match. We omit that for simplicity of presentation.

In this work, we mainly focus on *double haplotype match patterns* ($B = 2$) or *triple haplotype match patterns* ($B = 3$). We will also mainly focus on pair segment matching ($W = 2$), and single thread composite patterns ($w = 1$). We will present memory-efficient multi-column scanning PBWT-based algorithms: *double-PBWT* for identifying double haplotype match patterns, and *triple-PBWT* for triple haplotype match patterns.

3 Multi-column PBWT

In PBWT, a single column scans the haplotype panel from left to right and updates divergence values and prefix arrays to output long matches. Let $P_{k,L}$ signify the active single column of PBWT operating at site k finding matches of length at least L **ending** at site k. If the panel were to be scanned in a reverse fashion from right to left then, $R_{N-k,L}$ would signify the active single column of PBWT operating at site $N-k$ finding matches of length at least L ending at site $N-k$. This would allow the user to compare two sets of matches at site k (or $N-k$ if looking from right to left) i.e. matches from $P_{k,L}$ and $R_{N-k,L}$. This method would require the panel to be scanned twice if we wanted to compare such matches along different SNP sites of the panel. In fact, one approach would be to scan the panel in reverse and save all the precomputed divergence values and prefix arrays before hand as in [16]. While this method works well, it occupies disk-space and needs to be loaded into memory which might not be efficient for larger panels. In contrast, mcPBWT is capable of providing the same information on a single pass of the haplotype panel from left to right using multiple columns of PBWT without having to pre-scan the panel and save the values on disk.

The general idea of mcPBWT is to utilize the information obtained from "look-ahead" PBWT columns. Here, a new PBWT column operating at site $k+L$ that finds all the matches **starting** at site k is denoted by $P'_{k+L,L}$ such that matches found by columns P' and R are the same i.e. $P'_{k+L,L} = R_{N-k,L}$. So mcPBWT would consist of a set of PBWT columns where the columns are finding matches ending at a site or starting few sites before. Even though, we only talk about two columns of PBWT here, similar approach can be also used for the case of three columns of PBWT. In fact, we can spawn multiple such columns of PBWT and at each column, the user has the flexibility to find either matches ending at that site as in PBWT or have the flexibility to find matches starting certain number of sites before, depending on the use case. First, we discuss the divergence value properties that enable us to find the matches **starting** at a certain site and then the two specific cases of mcPBWT in the following sections.

3.1 Divergence Value Properties

There are two major properties of the divergence values that assist in finding the matches starting at a certain site efficiently. The first property asserts that in a divergence array, adjacent divergence values cannot be equal unless it is zero (or when a match does not exist)[1], while the second property that extends the first property asserts that between any two consecutive equal divergence values there must be a divergence value that is greater than those equal values. These properties are presented formally as two lemmas below.

Lemma 1. *Two adjacent divergence values aren't equal unless it is zero (or when the divergence value greater than current k, i.e. when there is no match).*

Proof. This property mentioned by Durbin, asserts that $d_k[i-1] \neq d_k[i]$, $0 < i < M$ except when $d_k[i] = 0$ or $d_k[i] = k$. For any index i, the divergence value at some site k, $d_k[i]$ gives the starting position of a match between haplotypes at index i and $i-1$. Since the panel is bi-allelic and reverse prefix sorted, this means that $y_{i-1}[d_k[i]-1]$ must be 0 and $y_i[d_k[i]-1]$ must be 1. The same condition holds for $d_k[i-1]$ in that $y_{i-2}[d_k[i-1]-1]$ must be 0 and $y_{i-1}[d_k[i-1]-1]$ must be 1. But if we assume that $d_k[i] = d_k[i-1]$, there is a contradiction on the value for $y_{i-1}[d_k[i]-1]$. This proves that the adjacent divergence values can't being equal unless the divergence value is equal to 0 or k (match does not exist). □

Lemma 2. *In a divergence array d_k, assume $d_k[i] = x$, and there exists another index $g, g < i$ and $g \neq i-1$ where g is the first index preceding i to have divergence value equal to x i.e. $d_k[g] = x$,*

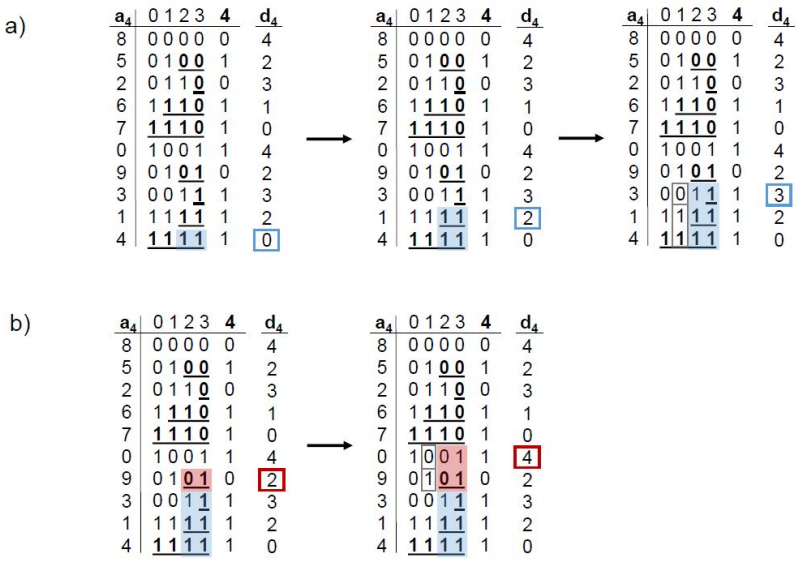

Figure 1: Two blocks of matches of minimum length 2 starting at site $k = 2$ in a panel of 10 haplotypes reverse prefix sorted at $k = 4$. The process of block detection while scanning the divergence array is shown. The rightmost colored rectangle (under d_4) shows the divergence value being scanned while the solid colored rectangles, blue in (a) and red in (b), show the actual matching blocks. The grey boxes at position $k = 1$ show the two groups within each block.

then there must be a divergence value greater than x at index h, where $g < h < i$ and $d_k[h] > x$ (except when $x = 0$) for $0 < g, h, i < M$.

Proof. This property is essentially an extension of lemma 1. In a bi-allelic panel, $d_k[i] = x$ asserts that $y_{i-1}[x-1] = 0$ and $y_i[x-1] = 1$. For index h in the range (g, i), the divergence value can either be $d_k[h] < x$ or $d_k[h] > x$ since g is the first index preceding i where $d_k[g] = x$. If we assume that all the divergence values in the range (g, i) are less than x , we can conclude that $y_h[x-1] = 0$, $\forall\ g < h < i$. This includes $d_k[g+1] < x$, which implies that $y_{g+1}[x-1] = y_g[x-1] = 0$. However, we already have the case that $d_k[g] = x$, which means that $y_g[x-1] = 1$ and $y_{g-1}[x-1] = 0$. That is a contradiction for the value of $y_g[x-1]$ which proves that it cannot be the case that all the divergence values are less than x. Hence, there must be a divergence value greater than x in the range (g, i). □

3.2 Finding blocks of starting matches

In a PBWT panel, neighboring haplotypes sharing matching segments cluster together. Such a collection of neighboring matches is called a *block*. Such blocks of matches are separated by a haplotype whose divergence value is greater than the difference of the site being observed and the length threshold specified. When iterating over the divergence array at a certain site, d_k , the divergence value properties discussed above restrict the combinations for the ordering of divergence values in the array. This in-turn assists in finding such blocks of matches where the matches share the same starting position.

Fig. 1 shows an example of using the divergence value properties to find blocks of starting matches of length, $L \geq 2$. It shows a reverse prefix sorted panel of 10 haplotypes at site $k = 4$,

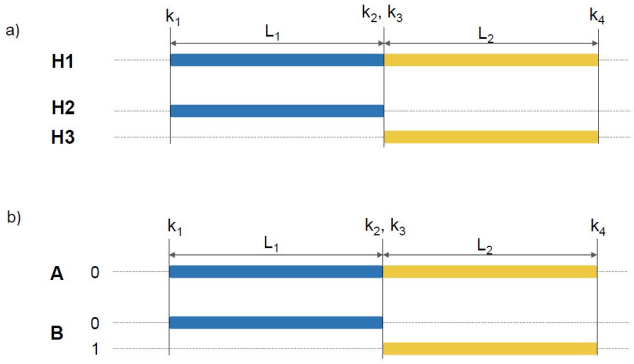

Figure 2: Double haplotype match patterns (a) Double haplotype match pattern where $c(p_0) = \{H1, H2\}$, $c(p_1) = \{H1, H3\}$ and $L(p_0) \geq L_1$, $L(p_1) \geq L_2$. Such formation represents crossover breakpoints. (b) Alternating match where A and B are two individuals and individual B has an alternating match with respect to individual A.

where d_4 shows the values of the divergence array and a_4 shows the prefix array. **Fig. 1(a)** shows the first block of starting matches found and **Fig. 1(b)** shows the second block found. These two blocks are separated by the sequence indexed 3 in a_4 with divergence value 3 which is greater than $k - L = 2$. The grey box at site $k = 1$ in the figures show the two clusters being formed within a block, called *groups*. Within each block, the top group consists of zeros at site $k - L - 1$ and the bottom group consists of ones at that site. It's also to be noted that a block of starting matches must contain only one sequence with divergence value equal to $k - L$ value. In the blue block, that sequence is 1, and in the red block, that sequence is 9. The haplotypes belonging to the same block but different groups constitute the actual matches that are of minimum length 2 and start at site $k = 2$. These starting matches are given by the tuples $(3, 1), (3, 4)$ for the blue block and $(0, 9)$ for the red block.

3.3 double-PBWT

double-PBWT, as the name suggests, uses two columns of PBWT simultaneously scanning the panel from left to right. The two columns are D_{k,L_1} and D'_{k',L_2} where column D_{k,L_1} is operating at site k finding matches of length at least L_1 ending at site k while column D'_{k',L_2} is operating at site k', L_2 sites ahead of k such that $k' = k + L_2$. This leading column finds matches of length at least L_2 starting from site k. Such a formulation of mcPBWT enables us to evaluate matches flanking on either side of a site. Such flanking matches can be defined by a composite haploytpe match pattern with $B = 2$, namely, double haplotype match pattern where $\mathcal{P} = \{p_0, p_1\}$, $|c(\mathcal{P})| = 1$, $W(p_0) = W(p_1) = 2$ and $g = 0$(for simplicity). This form of composite pattern is structured to signify patterns of recombination. Such combination of matching pair segments were also shown to be potential phasing errors[7]. Here, we introduce and focus on one such variation of this composite pattern termed alternating match but the algorithm can find all the other possible variations as well. We define alternating match and the problem statement formally in the following paragraphs.

Alternating Match Definition. An alternating match is a strict case of a double haplotype match pattern, $\mathcal{P} = \{p_0, p_1\}$, $|c(\mathcal{P})| = 1$, $W(p_0) = W(p_1) = 2$ where individual information is also encoded. If we assume the single thread haplotype belongs to an individual say, A, the non-thread haplotypes in p_0 and p_1 must be the complementary haplotypes of the same individual, say, B.

Such a case of double haplotype match pattern is termed as an *alternating match*.For instance, if p_0 is a match between $A_i[k_1, k_2)$ and $B_j[k_1, k_2)$where, $i, j \in \{0, 1\}$ and A_i, B_j are haplotypes of individuals A and B respectively and A_i is the thread haplotype. Then p_1 must be a match between $A_i[k_3, k_4)$ and $B_{j'}[k_3, k_4)$, where $B_{j'}$ is the complement haplotype of individual B. Here, B is said to have an alternating match with respect to A. **Fig. 2** shows the double haplotype match pattern and an alternating match. **Fig 2(a)** shows the double haplotype match pattern where $H1, H2$ and $H3$ are three haplotypes. Here, sequence id set of p_0, i.e., $c(p_0) = \{H1, H2\}$ and sequence id set of p_1, $c(p_1) = \{H1, H3\}$ where $c(p_0) \cap c(p_1) = \{H1\}$ is the thread haplotype. This form of double haplotype match represents crossover breakpoints. Similarly, **Fig. 2(b)** shows an example of an alternating match where individual B has an alternating match with respect to individual A. Here, sequence id set of p_0, i.e., $c(p_0) = \{A_0, B_0\}$ and sequence id set of p_1, $c(p_1) = \{A_0, B_1\}$ such that A_0 is the thread haplotype and B_0 and B_1 are complementary haplotypes of individual B. The first segment of the alternating match (the blue segment) is of length L_1 and the second yellow segment is of length L_2. **Fig. 2(b)** shows the case where the boundaries of the two pairs are juxtaposed such that $k_2 = k_3$, i.e. $g = 0$.

Problem Statement. Given a phased haplotype panel and user specified length parameters L_1, L_2 and g (here assumed 0 for simplicity), find all the alternating matches. We discuss the simple case of the alternating matches with strict boundaries as shown in **Fig. 2(b)** such that $g = 0$. The algorithm proceeds where the leading PBWT column D' first stores the information on starting matches using block and group properties, passes this information to the lagging PBWT column D which then checks if an alternating match exists for each ending match pair found of minimum length L_1. These two PBWT columns are L_2 ($k' - k = L_2$) distance apart so that the starting matches found are of length at least L_2.

Algorithm 1 shows the working mechanism for PBWT column D'. It updates the block and group information for all the starting matches found at a given site. The *block* array of size M keeps track of the block-membership of all the haplotypes, where the index of the array represents the haplotypes' indexes. An integer value (id) is assigned to haplotypes belonging to the same block. *group*, an array of size M distinguishes between the haplotypes of the block with 0 or 1 at $k' - L_2 - 1$ position. The index of this array also represents the haplotypes. *rblock* stores the same block and group information in a dictionary format where it is indexed by the block id to find the double haplotype match pattern show in **Fig. 2(a)**. The divergence values and prefix arrays are calculated using Durbin's algorithm 1 and 2 [1]. The *block* and *group* arrays along with *rblock* are updated simultaneously as the divergence and prefix arrays are updated so the time complexity to find the blocks of starting matches is $O(M)$ at each site and $O(MN)$ for the all sites. The block and group arrays are passed on to the PBWT column D where it decides if an alternating match is found. Here, the divergence array is scanned from $M - 1$ to 1 as it's more intuitive to understand the formation of block and groups but it can be scanned in the other direction without affecting the algorithm as shown later in Algorithm 4

Algorithm 2 is responsible for finding the other set of matches ending at site k and deciding if an alternating match exists. This algorithm scans the panel from site 0 to N simultaneously as D' scans the panel ahead of it. It finds all matches ending at site k using Durbin's algorithm 3 [1]. For every such ending match, it checks to see if there's a starting match that satisfies as an alternating match. This is done using the block and group arrays passed from D'. When the condition is met, the alternating match is reported. This checking is done in constant time since array access is constant time. Because of this constant time lookup for every pair of ending match found, the time complexity depends on the number of such match pairs found. We define C as the total number of

Algorithm 1 Leading PBWT at column D'_{k',L_2}: Find blocks and groups of matches starting at position k

create arrays $block[]$, $group[]$, $s[]$
create $rblock\{\}$ ▷ Store haplotypes belonging to blocks
$id \leftarrow 1$
$f \leftarrow true$
for $i = M - 1$ **to** 0 **do**
  if f **then** ▷ Start looking for a block
    if $d_{k'}[i] < k' - L_2$ **then**
      $s.add(a_{k'}[i])$
    else if $d_{k'}[i] == k' - L_2$ **then** ▷ Block exists
      $s.add(a_{k'}[i])$
      create $rblock\{id\} \leftarrow \{\}$
      for all $j \in s$ **do**
        $block[j] \leftarrow id$
        $group[j] \leftarrow 1$
        $rblock\{id\}.add(j)$
      $f \leftarrow false$, clear s
    else
      clear s
  else
    if $d_{k'}[i] < k' - L_2$ **then**
      $s.add(a_{k'}[i])$
    else ▷ Block ends
      $s.add(a_{k'}[i])$
      for all $j \in s$ **do**
        $block[j] \leftarrow id$
        $group[j] \leftarrow 0$
        $rblock\{id\}.add(j)$
      $id \leftarrow id + 1, f \leftarrow true$

match pairs found across all the sites. Therefore, the time complexity for this column is $O(C)$.

It is to be noted that this algorithm does not handle for the double haplotype match pattern in **Fig. 2(b)** but *rblock* can be used to query such patterns easily. For every ending match pair, like $H1$ and $H2$ detected by lagging PBWT at column D, the haplotypes belonging in the blocks of $H1$ and $H2$ are scanned using *block* and *rblock*, to find the second matching segment $H1$ and $H3$ ($H1$ being the thread haplotype) or $H2$ and $H3$ ($H2$ being the thread haplotype). This scanning process is done in $O(b)$ time for every ending match pair where b is the average number of haplotypes in a block. Hence, the overall time compleixty for such an algorithm would be $O(C * b)$ across all the sites.

Algorithm 3 shows the synchronous execution of both PBWT columns as it does a one pass scan on the panel. Since both columns move simultaneously across all the sites, the overall complexity of the algorithm is $O(MN + C)$. While we only show the case of alternating match when $g = 0$, these algorithms can be extended to handle for overlaps or gaps ($g \geq 1$) by altering the distance between the two PBWT columns.

Algorithm 2 Lagging PBWT at column D_{k,L_1}: Report matches of length at least L_1 forming alternating matches

```
block [], group [], rblock{}                                   ▷ Obtained from Algorithm 1
u ← 0 , v ← 0, create empty arrays a[],b[]                     ▷ Modified Durbin's algorithm 3
for i = 0 to M − 1 do
    if d_k[i] > k − L_1 then
        if u > 0 and v > 0 then
            for all 0 ≤ i_u < u and 0 ≤ i_v < v do
                for match from a[i_u] to b[i_v] ending at k_b do
                    if block[a[i_u]] = block[b[i_v]^c] then     ▷ b[i_v]^c is the complement of b[i_v]
                        if group[a[i_u]] ≠ group[b[i_v]^c] then
                            report alternating match
                    if block[b[i_v]] = block[a[i_u]^c] then
                        if group[b[i_v]] ≠ group[a[i_u]^c] then
                            report alternating match
        u ← 0, v ← 0
    if y_i[k] = 0 then
        a[u] ← a_k[i], u ← u + 1
    else
        b[v] ← a_k[i], v ← v + 1
```

Algorithm 3 *double-PBWT*: Simultaneous run of two PBWT columns

```
k ← 0, k' ← 0
while k' < N do
    run Algorithm 1                                            ▷ compute block[] and group[] for k'
    if k' − k ≥ L_2 then                                       ▷ feed block[] and group[] to algorithm 2
        run Algorithm 2                                        ▷ report alternating matches at k
        k ← k + 1
    k' ← k' + 1
```

3.4 double-PBWT: Comparing Block of Matches

So far we've used *double-PBWT* to find double haplotype composite matching patterns where $W(p_b) = 2$, $b = 0, 1$ but here we take advantage of its versatility to make comparisons between blocks of matches, i.e. $W(p_b) \geq 2$, $b = 0, 1$. The main idea here is to only evaluate matching blocks found by the two PBWT columns when they satisfy user-specified constraints of a valid block structure. A block structure is defined as a block consisting of at least $W^{'}$ haplotypes in common and sharing at least L long segments. This definition is adapted from cPBWT [13] and allows us to process composite match patterns in blocks. Such block-based comparison can be useful in studying recombination patterns too [16]. Here, both columns of PBWT store haplotypes that belong to different blocks and the blocks found by the two columns are compared to see if they share at least $W^{'}$ haplotypes. When this requirement is met, the blocks are output. While **Fig. 3** shows the comparison of blocks of haplotypes, this can also be extended to find alternating matches. Since alternating matches have more structure in terms of individuals that the haplotypes belong to, the algorithm needs to be modified to account for this constraint. For alternating matches in a block structure, additional constraints can be specified for the minimum number of thread haplotypes i.e. $|c(\mathcal{P})| \geq w_{min}$ and the minimum number of alternating individuals that should be present in a block-structure.

Fig. 3 shows an example of analyzing blocks of matches of length $L_1, L_2 \geq 2$ on either side of

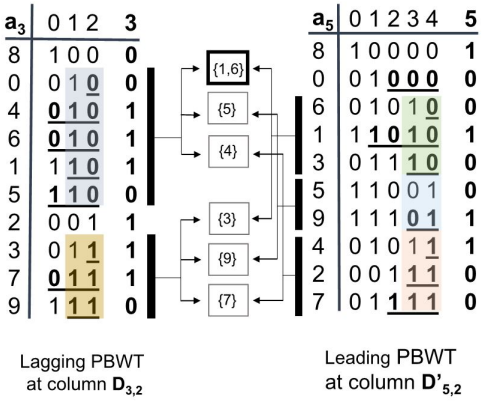

Figure 3: Double PBWT finding blocks of matches of length $L_1, L_2 \geq 2$ and $W^{'} \geq 2$. (Left) Haplotype panel with 10 haplotypes reverse prefix sorted at $k = 3$. The two colored boxes represent the block of matches ending at $k = 3$. (Right) Haplotype panel reverse prefix sorted at $k^{'} = 5$ where the colored boxes represent blocks of matches starting at $k = 3$. (Middle) The boxes in the middle show haplotypes common between the lagging and leading blocks. The grey block from lagging PBWT column $D_{3,2}$ and green block from leading PBWT column $D^{'}{}_{5,2}$ share two haplotypes $\{1, 6\}$ to satisfy $W^{'} \geq 2$ showing that the two haplotypes have an extending match.

site 3. Here, a valid block structure should have at least 2 haplotypes ($W^{'} \geq 2$). The only valid block structure found is shown in the middle with two haplotypes (6, 1) common to the two top blocks indicating that they share an extending match. It can be seen that this can be generalized to handle mismatches to study recombination patterns by adjusting the distance between the two PBWT columns.

3.5 triple-PBWT

triple-PBWT is the case of mcPBWT where three columns of PBWT are utilized. Each column has the freedom to find matches ending at those sites or starting few sites before. Here, we define *triple-PBWT* with three columns T_{k,L_1}, $T^{'}{}_{k',L_2}$ and $T^{''}{}_{k'',L_3}$ where, $k^{'} = k+L_2$ and $k^{''} = k+L_2+L_3$.

triple-PBWT can be useful in finding a triple haplotype composite match pattern ($B = 3$),

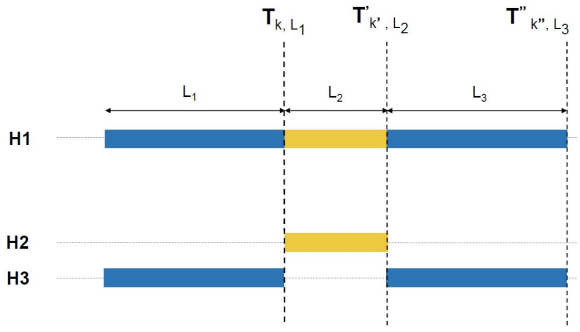

Figure 4: A composite pattern of three matching segments between three haplotypes $H1, H2$ and $H3$. This pattern is representative of gene conversion, where the yellow segment is a possible gene conversion tract such that $L_2 << L_1, L_3$. The vertical dashed lines show the three simultaneous PBWT runs each operating at sites $k, k^{'}$ and $k^{''}$, where $k^{'} = k + L_2$ and $k^{''} = k + L_2 + L_3$.

Algorithm 4 *triple-PBWT* at column $T^{'}$: Find matches that are of exact length L_2 at site $k^{'}$

create $block[], group[], s[]$ ▷ For queries of exact match pairs
create rblock{} ▷ Store haplotypes belonging to blocks
Create $end[]$ ▷ Tracks haplotypes that have 0 or 1 in the next site
$id \leftarrow 1, f \leftarrow False$
for $i = 0$ **to** $M - 1$ **do**
    if $d_{k'}[i] > k^{'} - L_2$ **then**
        if not $s.empty()$ **and** f **then**
            for $e \in s$ **do**
                $block[e] \leftarrow id$, $group[e] \leftarrow 1$
                $rblock\{id\}.add(e)$
            $id \leftarrow id + 1$
            $s.clear()$, $s.add(a_{k'}[i])$
            $f \leftarrow False$
        else
            $s.clear(), s.add(a_{k'}[i])$
    else if $d_{k'}[i] < k^{'} - L_2$ **then**
        $s.add(a_{k'}[i])$
    else if $d_{k'}[i] == k^{'} - L_2$ **then**
        $f \leftarrow True$
        if not $s.empty()$ **then**
            Create $rblock\{id\} \leftarrow \{\}$
            for $e \in s$ **do**
                $block[e] \leftarrow id, group[e] \leftarrow 0$
                $rblock\{id\}.add(e)$
            $s.clear()$, $s.add(a_{k'}[i])$
    if $y_i[k^{'}] = 0$ **then**
        $end[i] \leftarrow 1$ ▷ haplotype ends in 0
    else
        $end[i] \leftarrow -1$ ▷ haplotype ends in 1
if not $s.empty()$ **and** f **then** ▷ Boundary case
    for $e \in s$ **do**
        $block[e] \leftarrow id, group[e] \leftarrow 1$
        $rblock\{id\}.add(e)$

$\mathcal{P} = \{p_b\}$, $|c(\mathcal{P})| = 1$ and $W(p_b) = 2$, $b = 0...2$. The set of length constraints for the three matching pairs are $\{L_1, L_2, L_3\}$ such that $L(p_0) \geq L_1, L(p_1) = L_2$ and $L(p_2) \geq L_3$ and $L_2 << L_1, L_3$. Here, L_2 is restricted to be a short match in comparison to L_1 and L_3 to emulate a gene-conversion tract. **Fig. 4** shows an example of triple haplotype match pattern. Here, $H1, H2$ and $H3$ are three haplotypes where, $H1$ is the thread haplotype. Similarly, the sequence id sets are $c(p_0) = \{H1, H3\}$, $c(p_1) = \{H1, H2\}$ and $c(p_2) = \{H1, H3\}$. The three columns of *triple*-PBWT are represented by the vertical dashed lines. Here, column T runs a standard PBWT finding matches of length at least L_1 ending at site k. The $T^{'}$ column finds matches of exact length L_2 starting at site k and ending at site $k + L_2$ and column $T^{''}$ finds matches of length at least L_3 starting from site $k + L_2$. While a simple composite pattern like the one shown can be representative of gene-conversion tract, it's not sufficient condition and care has to be taken since the smaller match pair p_1 could end up providing lots of false positives. Additional information has to be incorporated to this formulation to distinguish false positives from true gene-conversion tracts but this shows one potential use case for the algorithm. The first column behaves similar to **Algorithm 1** and can be easily extended from there. The last column runs **Algorithm 2** as in *double-PBWT*. The only change is for PBWT column $T^{'}$. **Algorithm 4** shows how the exact matches can be catalogued for column $T^{'}$. Exact matches satisfy both restrictions of starting and ending matches and hence the algorithm uses ideas of both starting and ending matches to find them. Here, *block* and *group* arrays along with *rblock* serve the same function as in *double-PBWT*. *rblock* is a dictionary indexed by the block ids that store haplotype indices belonging to such blocks. This dictionary is used to find exact match pairs like p_1. A new array *end* is introduced which keeps track of whether the haplotypes have 0 or 1 in the next variant site. This *end* array is utilized to filter the exact matches from the starting matches. Since, the *end* array is updated along with divergence and prefix arrays, the time complexity for this algorithm at a given site is $O(M)$. The overall synchronization of the three columns is similar to *double-PBWT* in that columns $T^{'}$ and $T^{''}$ catalogue the exact matches and starting matches respectively and pass this information in the form of *block*, *group*, *rblock* and *end* (for column $T^{'}$) to column T. Then, for every ending match pair detected by column T, it takes constant time to look up match pairs p_2 but the haplotypes that belong to $H1$'s block have to be scanned for match pair p_1. When those matching pair segments exist, the triple haplotype match is reported.

Of the three PBWT columns of *triple-PBWT*, column $T^{''}$ is the same as the leading column of *double-PBWT* and hence has a time complexity of $O(M)$ at each site and $O(MN)$ across all variant sites. For the middle column $T^{'}$, the time complexity to catalogue the exact matches is $O(M)$ at each site and $O(MN)$ across all the sites. It's important to note that querying of exact matches of length L_2 can be done by accessing $block, group, end$ arrays in constant time. Lastly, column T makes constant time query for every ending match found to see if the last match pair exists but has a time complexity of $O(b)$ to find the middle match pair, where b is average number of haplotypes in a block. Since, T is our trigger column, the time complexity depends on the number of match pairs found by PBWT column T. Hence, we define the complexity of this column across all sites similar to *double-PBWT* as $O(C * b)$, where C is the total number of match pairs found across all the variant sites. The overall time complexity of Triple PBWT is then $O(C * b + MN + MN)$, i.e. $O(MN + C * b)$.

4 Discussion

In this work, we present a more flexible and powerful variation of PBWT for detecting composite haplotype matches. The original formulation in PBWT for the haplotype matching problem only

captures the matching pattern at every single column separately. Our algorithm, however, simultaneously captures the patterns across multiple columns of PBWT. In the single column matching formulation, at the active column of the PBWT, one only has access to the information in the past but is uninformed about future columns. In our formulation, the columns at the forefront can provide "look ahead" information allowing the algorithm to make complex decisions. Our flexible algorithm's capability to analyze composite matching patterns opens new potentials of the PBWT data structure.

The proposed method does not require to output or book keep the matches which will be very useful in analyzing large haplotype panels with millions of individuals. While the PBWT algorithm is able to find all matches efficiently, the number of matches in large cohorts may be enormous. As a result, analyzing composite patterns like alternating matches after the PBWT run may not be very efficient. Hence, a flexible algorithm like *double-PBWT* would be useful in such cases. Additionally, we also showed that the *triple-PBWT* could find composite haplotype match representative of gene-conversion tracts. This shows that mcPBWT has potential to allow and adjust for flexible matching criteria and is suitable for more general-purpose settings.The double haplotype match pattern discussed where $H1$ is the thread haplotype and $H2$ and $H3$ have matches with it not only identifies more recombination events but also provides plausible evidence that $H2$ and $H3$ coalesce more recently. This could help to determine the time of the recombination events, and also help "triangulating" the genealogical relationship among individuals carrying these matching segments. Such analyses can also be conducted using blocks to enable stronger signal using mcPBWT.

Acknowledgements

PS, AN, DZ and SZ were supported by the National Institutes of Health grant R01 HG010086. AN, DZ and SZ were also supported by the National Institutes of Health grants R56 HG011509. AN and DZ were also supported by the National Institutes of Health grant OT2-OD002751.

References

[1] Durbin, R. Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). *Bioinformatics* **30**, 1266–1272 (2014).

[2] Loh, P.-R. *et al.* Reference-based phasing using the haplotype reference consortium panel. *Nature Genetics* **48**, 1443 (2016).

[3] Delaneau, O., Zagury, J.-F., Robinson, M. R., Marchini, J. L. & Dermitzakis, E. T. Accurate, scalable and integrative haplotype estimation. *Nature Communications* **10**, 1–10 (2019).

[4] Rubinacci, S., Delaneau, O. & Marchini, J. Genotype imputation using the positional burrows wheeler transform. *bioRxiv* 797944 (2020).

[5] Naseri, A., Liu, X., Tang, K., Zhang, S. & Zhi, D. RaPID: Ultra-fast, powerful, and accurate detection of segments identical by descent (IBD) in biobank-scale cohorts. *Genome Biology* **20** (2019).

[6] Zhou, Y., Browning, S. R. & Browning, B. L. A fast and simple method for detecting identity-by-descent segments in large-scale data. *The American Journal of Human Genetics* **106**, 426–437 (2020).

[7] Freyman, W. A. *et al.* Fast and robust identity-by-descent inference with the templated positional burrows–wheeler transform. *Molecular Biology and Evolution* **38**, 2131–2151 (2021).

[8] Thompson, E. A. Identity by descent: Variation in meiosis, across genomes, and in populations **194**, 301–326 (2013).

[9] Novak, A. M., Garrison, E. & Paten, B. A graph extension of the positional burrows–wheeler transform and its applications. *Algorithms for Molecular Biology* **12**, 1–12 (2017).

[10] Sanaullah, A., Zhi, D. & Zhang, S. d-PBWT: dynamic positional Burrows-Wheeler transform. In *International Conference on Research in Computational Molecular Biology*, 269–270 (Springer, 2020).

[11] Naseri, A., Zhi, D. & Zhang, S. Multi-allelic positional burrows-wheeler transform. *BMC Bioinformatics* **20**, 1–8 (2019).

[12] Williams, L. & Mumey, B. Maximal perfect haplotype blocks with wildcards. *Iscience* **23**, 101149 (2020).

[13] Naseri, A., Zhi, D. & Zhang, S. Discovery of runs-of-homozygosity diplotype clusters and their associations with diseases in uk biobank. *medRxiv* (2020). Doi: 10.1101/2020.10.26.20220004.

[14] Alanko, J., Bannai, H., Cazaux, B., Peterlongo, P. & Stoye, J. Finding all maximal perfect haplotype blocks in linear time. *Algorithms for Molecular Biology* **15**, 1–7 (2020).

[15] Cunha, L., Diekmann, Y., Kowada, L. & Stoye, J. Identifying maximal perfect haplotype blocks. In *Brazilian Symposium on Bioinformatics*, 26–37 (Springer, 2018).

[16] Naseri, A., Yue, W., Zhang, S. & Zhi, D. Efficient Haplotype Block Matching in Bi-Directional PBWT. In Carbone, A. & El-Kebir, M. (eds.) *21st International Workshop on Algorithms in Bioinformatics (WABI 2021)*, vol. 201 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 19:1–19:13 (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2021).