# Efficient Haplotype Block Matching in Bi-Directional PBWT

**Ardalan Naseri**[1]
School of Biomedical Informatics, University of Texas Health Science Center at Houston, TX, USA

**William Yue**[1]
School of Biomedical Informatics, University of Texas Health Science Center at Houston, TX, USA

**Shaojie Zhang**[2]
Department of Computer Science, University of Central Florida, Orlando, FL, USA

**Degui Zhi**[2]
School of Biomedical Informatics, University of Texas Health Science Center at Houston, TX, USA

## Abstract

Efficient haplotype matching search is of great interest when large genotyped cohorts are becoming available. Positional Burrows-Wheeler Transform (PBWT) enables efficient searching for blocks of haplotype matches. However, existing efficient PBWT algorithms sweep across the haplotype panel from left to right, capturing all exact matches. As a result, PBWT does not account for mismatches. It is also not easy to investigate the patterns of changes between the matching blocks. Here, we present an extension to PBWT, called bi-directional PBWT that allows the information about the blocks of matches to be present at both sides of each site. We also present a set of algorithms to efficiently merge the matching blocks or examine the patterns of changes on both sides of each site. The time complexity of the algorithms to find and merge matching blocks using bi-directional PBWT is linear to the input size.

Using real data from the UK Biobank, we demonstrate the run time and memory efficiency of our algorithms. More importantly, our algorithms can identify more blocks by enabling tolerance of mismatches. Moreover, by using mutual information (MI) between the forward and the reverse PBWT matching block sets as a measure of haplotype consistency, we found the MI derived from European samples in the 1000 Genomes Project is highly correlated (Spearman correlation r=0.87) with the deCODE recombination map.

**2012 ACM Subject Classification** Applied computing → Genetics; Applied computing → Computational genomics

**Keywords and phrases** PBWT, Bi-directional, Haplotype Matching

**Digital Object Identifier** 10.4230/LIPIcs.WABI.2021.19

**Supplementary Material** *Software (Source Code)*: `https://github.com/ZhiGroup/bi-PBWT` archived at `swh:1:dir:ac5c1bfc4d8e31d338b887dd9d5131bbeefbe09a`

**Funding** This work was supported by the National Institutes of Health R01HG010086.

**Acknowledgements** This research has been conducted using the UK Biobank Resource under Application Number 24247.

## 1 Introduction

Diploid organisms such as humans inherit two copies of chromosomes, one from each parent. Each haplotype sequence of the two copies of a chromosome can be represented as a long string. Haplotype matches are usually considered to be binary and the matches between any pair of haplotypes may be due to a common ancestor or natural selection. While short

[1] These authors contributed equally to this work.
[2] Corresponding authors.


matches between two individuals may have been caused by chance, a match shared between several haplotypes or individuals are more informative since it is less likely that the matches occurred by random. The lower the probability of haplotype matches by chance, the stronger the evidence that the segments have been inherited from a common ancestor [14].

The availability of large-scale genetic data has promoted the need for efficient algorithms and methods. Positional Burrows-Wheeler Transform (PBWT) [5] proposed by Richard Durbin, provides an efficient data structure for compression and searching for matches in large haplotype sequences. The underlying idea of PBWT is to sort the haplotype sequences at each variant site $k$ by their reversed prefix order. An additional array, called the divergence array, keeps track of the reverse prefix match between adjacent haplotypes in sorted order. The presented algorithms in PBWT provide an efficient approach to find all pairwise matches between haplotype sequences ending at each variant site.

PBWT data structure has already been utilized to find matching blocks in large haplotype panels [4, 1, 11]. A matching block contains several haplotype sequences that share a common reverse prefix of length $L$ at a site. The concept of a matching block is a direct extension of pairwise matches to multi-way matches due to transitivity. Instead of a sequence with a minimum length of $L$ shared by only two haplotypes, a block of matches is defined by a minimum number of haplotypes or individuals ($W$), where $W \geq 2$.

For the sake of memory efficiency, the PBWT algorithms are typically a scan across a haplotype panel from the first variant site in a forward direction. At each variant site during the pass, the panel is sorted by the reversed prefix order while updating the divergence values for each haplotype. The computed values are only kept transiently and only the active array at the current variant site $k$ is kept in memory. As a result, the memory usage for PBWT algorithms will be only $O(M)$, where $M$ denotes the number of haplotypes. This, however, comes with the cost of only being able to access data at a single site and only from one direction. While the forward-only PBWT sweep is sufficient for identifying pairwise matches between any two haplotypes or matching blocks, it is not efficient in determining the boundaries of the matching blocks nor does it facilitate the tracking of changes in matching blocks. As a result, no mismatch will be allowed in otherwise long matching blocks and recombination events cannot be observed between blocks of haplotypes around a site.

Here, we present an extension to Durbin's PBWT by allowing the information at both directions from site $k$ to be accessible. This is enabled by a memory-efficient two-pass sweep: first, a reverse pass storing all PBWT data structures in an intermediate file and then a forward pass creating a transient forward PBWT. In the forward PBWT, we have random access to the reverse PBWT at each variant site and observe the potential changes beyond the variant site using the pre-computed reverse PBWT.

The bi-directional PBWT can be used to efficiently track haplotype sequences across the matching blocks at both the forward and the reverse PBWTs. Since the matching blocks at either side of a variant site define the partition of the sequence indices, the correspondence between the forward matching blocks and the reverse matching blocks can be expressed as the intersection of their partitions. In the context of PBWT, we formulate the problem as the *all partition matching problem*, where all matching partitions are enumerated, and the *large partition matching problem*, where only sufficiently large partitions are reported. Both problems can be solved by efficient $O(M)$ algorithms. In doing so, bi-directional PBWT allows integration of the changes within a matching block.

We demonstrate two potential applications of the haplotype block matching algorithm using bi-directional PBWT. The first application involves the identification of matching blocks allowing mismatches. Additionally, the blocks of matches that undergo changes around a site may indicate phasing errors or recombination events. Preliminary results of these applications are shown using UK Biobank and 1000 Genomes project data [3].

# 2 Methods

## 2.1 Background and notation

Following Durbin's notation [5], we define a panel of haplotype sequences as $X \in \{0,1\}^{M\times N}$, a two-dimensional matrix of binary values, where $M$ denotes the number of haplotype sequences and $N$ the number of variant sites. Each row is a haplotype sequence and each column $X_k, k = 0, ..., N-1$ is an array representing the values of haplotypes at the site $k$. If $s$ and $t$ are two haplotype sequences, and $s[j,k)$ and $t[j,k)$ denote their subsequences from site $j$ to site $k-1$, then there is a match between $s$ and $t$ from $j$ to $k$ if $s[j,k) = t[j,k)$.

The concept of haplotype match can be defined over blocks [4, 1, 11]. Assume $C$ is the set of sequence indices: $C = \{0, ..., M-1\}$. We can define a haplotype matching block or a block as a tuple $(c, j, k), j < k$, where $c \subset C$ is a subset of sequence indices and $j$ and $k-1$ denote the starting and ending sites of the match, respectively. The length of a block is defined as $l = k - j$, and the width of a block is $w = |c|$. If $l \geq L$ and $w \geq W$, we call the block a ($L$, $W$)-block. In particular, a block is called width maximal at each site $k$ if the number of sequences sharing a substring with a length $\geq L$ cannot be increased. A block is called length maximal if one or multiple sequences in the block have a different value at the site $k$ resulting in less than $W$ haplotypes in the block. Width and length maximal blocks can be found in $O(NM)$ time in Positional Burrows-Wheeler Transform (PBWT) [4, 1, 11].

The basic idea using PBWT to find matching blocks is that at any site $k$, all matching haplotype sequences with a length $\geq L$ are adjacent to each other in the PBWT panel [11]. In other words, if a set of haplotypes builds a block of matches with a length $L$, they will be contiguous in the PBWT panel.

### 2.1.1 PBWT data structures: PBWT matrix, positional prefix array, and divergence array

The underlying idea of the Positional Burrows-Wheeler Transform (PBWT) is to store haplotype sequences based on their reversed prefix order. It also enables efficient algorithms for identifying matches by introducing additional data structures augmenting the original haplotype panel $X$. The positional prefix array $a_k$ contains the sequence indices sorted based on their reversed prefix order. The PBWT array $y_k$ stores the values of haplotype sequences in the reversed prefix order at each site ($y_k[i] = X_k[a_k[i]]$). The divergence array $d_k$ at the variant site $k$ for each haplotype stores the starting position of the match between the haplotype with its preceding haplotype sequence in the reversed prefix order. In other words, the divergence value keeps track of the starting site index of the longest match for each haplotype. We refer to the value of the divergence array for each haplotype as its divergence value. $d_k$ is utilized to both identify a long match with a length $\geq L$ and determine the starting position of the match.

Note that in a typical PBWT all-versus-all matching algorithm, the haplotype panel is swept from left to right and the data structures $a_k$, $y_k$, and $d_k$ can be built transiently, thus enabling an efficient memory footprint of $O(M)$. Indeed this is the primary mode that PBWT can be used in large data sets. However, such sweeping algorithm precludes access to information from both sides of the site $k$, and thus limiting the investigation of changes of haplotype blocks.

## 2.2 Bi-directional PBWT

### 2.2.1 Overview

Here, we assume that PBWT data structures for both forward and reverse directions are available at each site. Figure 1 shows the bi-directional PBWT data structures using a simple example of a haplotype panel. Given a minimum length $L$, width maximal blocks ending at site $k$ define a partition of the set of indices $C$ for either side. Assume the *block set* $S(k, L)$ represents a set of blocks of matching haplotypes in the forward PBWT ending at a variant site $k$. In other words, $S(k, L)$ is a partition of the set of sequence indices $C$. Similarly, we define the block set $T(k, L)$ for the reverse PBWT at site $k$, another partition of $C$. For the sake of simplicity, we omit the $k$ and $L$ and denotes the partitions as $S$ and $T$. It is of our interest to find the matching between these two block sets, where the matching is defined over all subsets shared by $S$ and $T$, also known as the intersection of partitions $U(S, T) = \{s \cap t | s \in S \wedge t \in T\} \backslash \{\varnothing\}$. Additionally, the forward and the reverse PBWTs do not have to be back-to-back and a small gap may be allowed. I.e., we can consider the matching between $S(k, L)$ and $T(k + g, L)$, where $g$ is the size of the gap.

Here, we define two versions of the matching problems, the *all-block matching problem* and the *W-width block matching problem*. For the *all-block matching problem*, the goal is to enumerate all possible blocks in $U(S, T)$. In the case of haplotype sequences from population genotyping data, we expect there will be a strong correlation between the blocks of $S$ and $T$. This can be quantified by the mutual information (MI) between the sets of matching blocks in forward ($S$) and reverse ($T$) directions:

$$MI(S,T) = \sum_{s \in S} \sum_{t \in T} p_{(S \cdot T)}(s,t) log\left(\frac{p_{(S \cdot T)}(s,t)}{p_S(s) p_T(t)}\right),$$

where $p_{(S \cdot T)}(s, t)$ is the joint probability function of $S$ and $T$, which can be calculated by enumerating the blocks in $U(S, T)$.

For the *W-width block matching problem*, the goal is the find all the matching blocks in $U(S, T)$ that contain $\geq W$ haplotypes. This may sound like yet another way of solving the $(2L + g, W)$-block problem. However, the benefit of defining the problem as such is that it would allow mismatches in the gap region of an otherwise exact matching block.

Our approach for finding blocks of matches using bi-directional PBWT can be split into three subroutine algorithms: 1) Finding blocks of matches exceeding the given length and width cut-off using bi-directional PBWT. 2) Matching blocks from both sides. 3) Extracting the length of matches, in the case of finding $(2L + g, W)$-blocks. Figure 1 shows a simple schematic of our method for detecting clusters around the site $k$. The matching blocks from both sides are considered around a given gap in terms of sites.

### 2.2.2 Algorithm 1: Find width maximal blocks

We start by independently finding width maximal blocks ($(W, L)$-blocks) on both sides at each site $k$ that contain at least $W$ sequences and include at least $L$ sites. This will output the block sets $S(k, L)$ and $T(k, L)$ for each site $k$. Using the divergence arrays for each side, we can easily find blocks that extend for at least $L$ matching sites. Blocks form contiguous segments in the positional prefix array and are capped off at both ends by large divergence values. Blocks of matches are separated by a sequence $j$ where $d_j > k - L$. We can separate blocks of matches at each site and assign a unique ID to each block. For each sequence, we store its corresponding block ID in an array for lookup in Algorithm 2. Algorithm 1 has a

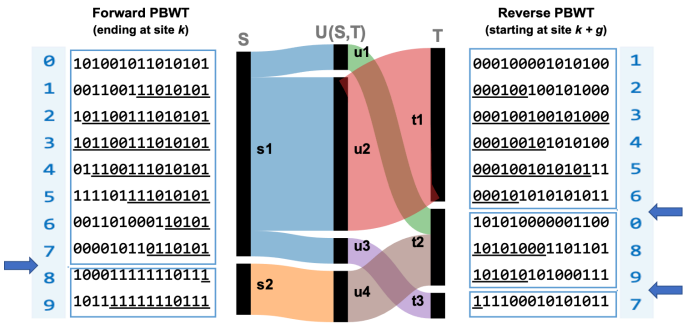


**Figure 1** Bi-directional PBWT data structures. The standard PBWT data structures for the forward and the reverse PBWTs are shown at both sides. For $L = 5$, the $L$-block set at the forward PBWT, $S = \{s1 = \{0, 1, 2, 3, 4, 5, 6, 7\}, s2 = \{8, 9\}\}$ and the $L$-block set at the reverse PBWT, $T = \{t1 = \{1, 2, 3, 4, 5, 6\}, t2 = \{0, 8, 9\}, t3 = \{7\}\}$, are integrated to form $U = U(S, T) = \{u1 = \{0\}, u2 = \{1, 2, 3, 4, 5, 6\}, u3 = \{7\}, u4 = \{8, 9\}\}$. The arrows on the sides mark the boundaries of matching blocks.

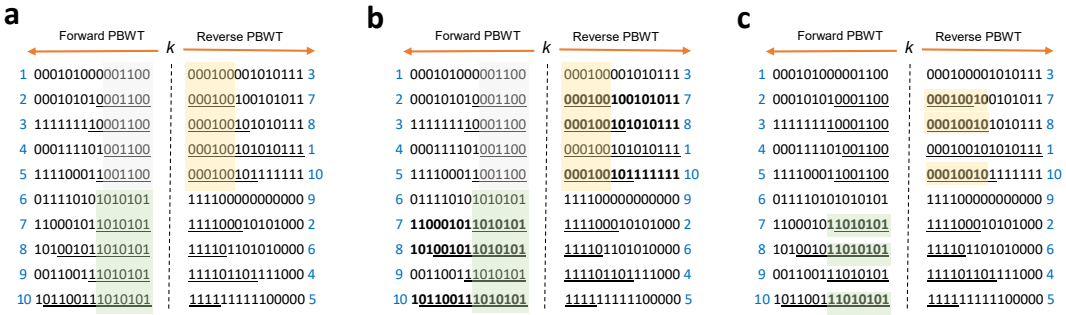


**Figure 2** A simple schematic of bi-directional PBWT for finding blocks of matches before and after the site $k$: 1. Finding blocks of matches exceeding a given length in terms of sites $L$ ($\geq 6$) and haplotypes $W$ ($\geq 3$) at both sides separately (a). 2. Merging the blocks of matches from both sides (b). 3. Determination of the length in terms of sites ($L$) of continuous matching blocks (c). The length in terms of the number of sites in the matching block in green increases since the 6th sequence is discarded.

time complexity of $O(M)$ since we only iterate through the divergence array at each site. Figure 2 illustrates the process for finding width maximal blocks and Algorithm 1 outlines the pseudo-code for finding width maximal blocks on one side of the site.

### 2.2.3 Algorithm 2: Match blocks by enumerating intersection of partitions

Recall from Algorithm 1 that width maximal blocks represent a set of sequences that match for at least $L$ sites. In Algorithm 2, we solve the *W-width block matching problem* by enumerating the intersection of block sets that exceed the minimum number of haplotypes ($W$) on both sides. Note that the *all-block matching problem* can be solved similarly by simply ignoring the minimum number of haplotypes $W$.

We define an array *link* where $link[i]$ is the pair <forward block ID of haplotype $i$, reverse block ID of haplotype $i$>. After sorting *link*, haplotypes in the same forward and reverse blocks will be adjacent to each other and form contiguous sections in the array. Each section in the array that has the same two-block pair becomes a candidate for a matching block

**Algorithm 1** Find width maximal blocks at site $k$.

```
blockID ← 1
start ← −1
for i ← 0, M − 1 do
    if d[i] > k − L then
        assign blockID to sequences in [start, i − 1]
        blockID = blockID + 1
        start ← i
    end if
end for
```

with the only criteria left to check being if the size of the block candidate is greater than or equal to $W$. Note that since the pairs in the *link* array only contain block ID values from $1 - M$, we can utilize a radix sort. Algorithm 2 has a time complexity of $O(M)$ time due to the radix sort and a memory complexity of $O(M)$. Figure 2 illustrates the process for merging blocks and Algorithm 2 outlines the pseudo-code for merging blocks.

**Algorithm 2** Match blocks in forward and reverse PBWT at position $k$.

```
link ← []
for i ← 0, M − 1 do
    link[i] ← <forward block ID of haplotype i, reverse block ID of haplotype i>
end for
radixSort(link)
start ← 0
for i ← 1, M − 1 do
    if link[i] ≠ link[i − 1] then
        if i − start ≥ W then                ▷ Check if the block's width is at least W
            report width-maximal matching block from [start, i − 1]
        end if
        start ← i
    end if
end for
```

### 2.2.4 Algorithm 3: Report block length

Recall that Algorithm 1 only checks that a block matches for a minimum of $L$ sites and doesn't compute how much further the block matches past those $L$ sites. Once a block candidate from Algorithm 2 has met the size requirement of containing at least $W$ sequences, we utilize Algorithm 3 to find the length of the block on both sides of site $k$. If on the left side, a block includes sequences at positions $x_0, x_1, ..., x_j$ in the positional prefix array, then we define *minPos* as the $\min_{0 \le i \le j} x_i$ and the *maxPos* as the $\max_{0 \le i \le j} x_i$. The length that the block extends to the left can then be represented by $k - \max_{minPos < i \le maxPos} d_i$. The same logic can be applied to find the length of the block on the right side. Note that we do not include *minPos* in the query range since the definition of the divergence array states that $d_i$ is a comparison between string $i$ and string $i - 1$ in the positional prefix array. To be able to compute the expression $\max_{minPos < i \le maxPos} d_i$ efficiently, we can utilize a range query data structure. Since we do not need to perform range updates, sparse table

provides a lightweight data structure that can be built in $O(M)$ time by dividing the data into blocks of size $b$ ($b \in \Theta(log(M))$) and computing the sparse table over the maximums in each block. For each query, there will be a maximum of two $b$-sized blocks that will not be completely encapsulated by the range query. These two $b$-sized blocks will be found at the two boundaries of the range query and the maximum for these two blocks can be computed manually and efficiently using bitwise operations. Each range query can be performed in $O(b)$ where $b$ is a small constant factor. Alternatively, Cartesian trees can be used which guarantees the time complexity of $O(M)$ for pre-processing and $O(1)$ for range queries. In practice, however, we found that using sparse tables with the block technique seemed to be faster than Cartesian trees in both precomputation and query runtimes. Since each sequence is only included in a single query and queries take $O(1)$ time, it takes $O(M)$ time to complete all queries. Thus, Algorithm 3's runtime is dominated by building the sparse table, which takes $O(M)$ time and $O(M)$ memory to construct. Figure 3 illustrates the process for reporting block length and Algorithm 3 outlines the pseudo-code for finding the length of a block reported in Algorithm 2.

**Algorithm 3** Report block length.

1: **for all** blocks reported in Algorithm 2 **do**
2: $l_f = k - \max_{minPos<i<=maxPos} d_f[i]$ ▷ report forward length
3: $k' = k + g - 1$
4: $l_r = \max_{minPos<i<=maxPos} d_r[i] - k'$ ▷ report reverse length
5: **end for**

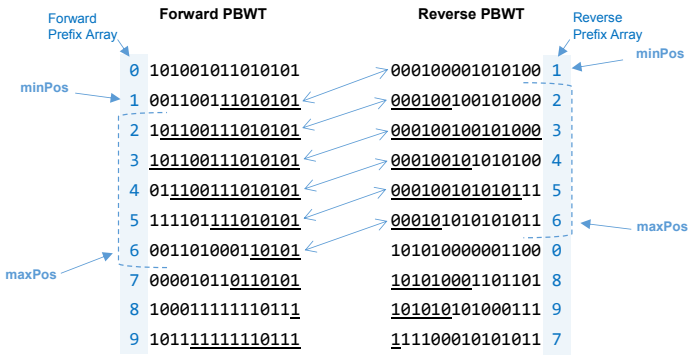


**Figure 3** A simple example illustrating Algorithm 3 – Report Cluster Length. To find the length of the cluster on both sides, we identify $minPos$ and $maxPos$ on both sides and perform a maximum range query on the divergence arrays.

### 2.2.5 Efficient implementation of bi-directional PBWT

To compute a bi-directional PBWT efficiently, we must be able to efficiently read a VCF file in reverse order (from the last site to the first site). In our implementation, we utilized the C++ method $seekg()$ to manipulate the position of the input stream pointer. We begin by reading the first site in the VCF file and extracting the number of individuals, $M$, by counting the number of "|". We then jump to the end of the file and start the process of

reading the VCF file in reverse order line by line. To extract each line from the end of the file, we use $seekg()$ to move the input stream pointer back one position at a time until we reach a newline character. Since we already know $M$, we can optimize this process by initially moving the input stream pointer back $4 * M$ positions since we know each individual in the VCF file takes up 4 characters. Then we only need to move the input stream pointer back one character at a time through the fixed fields which usually comprise less than 500 characters. By doing this, we can compute the reverse PBWT with comparable efficiency to the forward PBWT.

Bi-directional PBWT builds off of Durbin's algorithms 1 and 2. In these 2 algorithms, Durbin iterates through all $N$ sites while maintaining a positional prefix array (i.e. a sorted array of binary strings) and a divergence array ($d_i$ stores the smallest value $j$ such that the haplotype sequences in the prefix array positions $i$ and $i-1$ match from $site_j$ to the current site). Before running the aforementioned algorithms, we create the PBWT data structure for reverse sequences first and then use the stored values while scanning the panel in the forward direction. Note that we only need to store values from the reverse PBWT since we can simultaneously compute blocks and the forward PBWT at the same time. Source code is available at `https://github.com/ZhiGroup/bi-PBWT`.

# 3 Results

## 3.1 Runtime and memory usage

We evaluated the run time and memory usage of bi-directional PBWT on large-scale haplotype panels. The scalability of bi-directional PBWT was evaluated with respect to the size of the input data and the length parameter. Table 1 shows the resource consumption when running bi-directional PBWT (both forward and reverse combined) on chromosome 21 of the UK Biobank panel comprising 974,818 haplotypes with the parameters of $L = 0.5$ Mbps and $W = 100$. Table 2 shows the runtime growth of bi-directional PBWT with respect to the input size $M$ when all other variables are held constant. From Table 2, we can observe that our method has an asymptotic runtime of $O(M)$ with respect to $M$. Table 3 shows the runtime growth of bi-directional PBWT with respect to input size $N$ when all other variables are held constant. From Table 3, we can see that the algorithm has an asymptotic runtime of $O(N)$ with respect to $N$. Table 4 shows the runtime growth of bi-directional PBWT with respect to the parameter $L$ when all other variables are held constant. From Table 4, we can see that the asymptotic runtime of bi-directional PBWT is not affected by $L$. All experiments were carried out with a 2.10 GHz Intel Xeon E5-2620 v4 using a single core.

**Table 1** Run time and memory usage of bi-directional PBWT on chromosome 21 of the UK Biobank.

| Run time | Peak memory | Peak disk usage | #Blocks |
|---|---|---|---|
| 3.2 hrs | 261.44 MB | 73.1 GB | 245,563 |

$W = 100$, $L = 0.5$ Mbps.

**Table 2** Runtime growth with respect to the number of haplotypes $M$.

| Size of M | 50,000 | 100,000 | 200,000 | 400,000 | 800,000 |
|---|---|---|---|---|---|
| Time (s) | 607.1 | 1,432.54 | 3,000.11 | 5,971.03 | 11,284.42 |

$N = 9793$, $W = 100$, $L = 0.5$ Mbps.

**Table 3** Runtime growth with respect to the number of sites $N$.

| Size of N | 500 | 1000 | 2000 | 4000 | 8000 |
|---|---|---|---|---|---|
| Time (s) | 773.91 | 1,455.34 | 3,273.91 | 5,955.54 | 11,953.19 |

$M = 974818$, $W = 100$, $L = 0.5$ Mbps.

**Table 4** Runtime growth with respect to the target length $L$.

| Size of L (Mbps) | 0.1 | 0.2 | 0.4 | 0.8 | 1.6 |
|---|---|---|---|---|---|
| Time (s) | 12,090.35 | 12,336.94 | 13,727.56 | 13,471.45 | 13,957.97 |

$M = 974818$, $N = 9793$

The benchmarks show that the asymptotic runtime is linear with respect to the size of the input and that the asymptotic memory usage is $O(M)$.

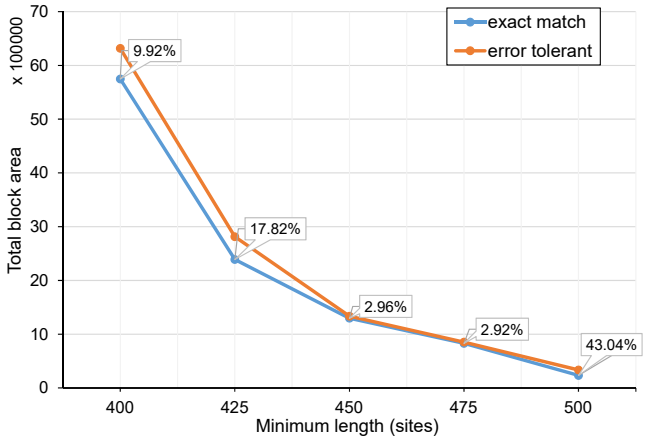


**Figure 4** Comparison of total areas covered by matching blocks using exact matches vs. allowing a mismatching site. The percentage increase in the total area covered by matching blocks is shown for different minimum target lengths.

## 3.2 Blocks of haplotype matches in UK Biobank

To establish the benefit of allowing mismatches in the otherwise perfect matching blocks, we ran bi-directional PBWT either with a tolerance of mismatch or with a perfect match in the gap region. We used a gap size of $g = 1$ and haplotypes of chromosome 21 in the UK Biobank data [2] to count the total sizes of matching blocks as the total area that is covered by any matching block. We used the total area to investigate the differences between exact matches and allowing only one mismatch (error) using bi-directional PBWT. The total area and the percentages of differences for exact matching blocks and error-tolerant mode are shown in Figure 4. The minimum number of haplotypes $W$ was set to 200. As shown in the figure, based on different cut-off values for the minimum length, the gain can vary from $\sim 3 - 43\%$. The percentage of additional area covered by matching blocks while allowing only 1 mismatch in the middle of blocks can result in $\sim 43\%$ more area compared to exact matches for $L = 500$.

We searched for haplotype matches using bi-directional PBWT in UK Biobank with a minimum length of 0.5 Mbps shared among at least 100 haplotypes using all individuals. Figure 5 shows the number of clusters at each site. Please note that there must be a mismatch

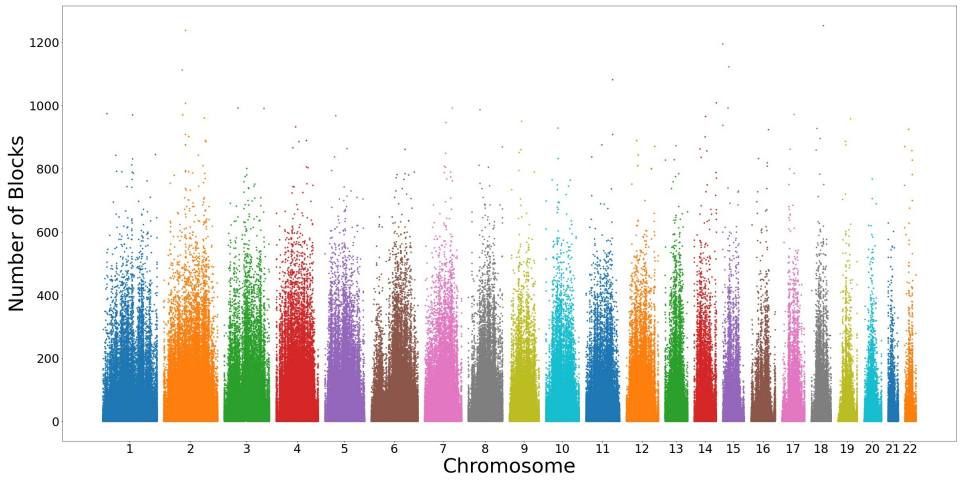


**Figure 5** Manhattan plot for the number of blocks allowing a mismatching site in the block using all autosomal chromosomes in the UK Biobank data. The minimum length for both sides was set to 0.5 Mbps.

at each site in the gap for the cluster to be counted. The minimum length should also hold for both sides of matches around each site. The plot shows the availability of such haplotype blocks across the chromosomes. There's also a dip in chromosome 6 which covers the HLA region. Figure 6 shows the average number of haplotypes that have an alternating allele in each block at every site. As shown in the figure, there is a peak in chromosome 6 which again comprises the HLA region. The mismatches in each cluster are shared by several haplotypes and the sizes of the blocks in terms of the number of haplotypes are larger compared to other regions.

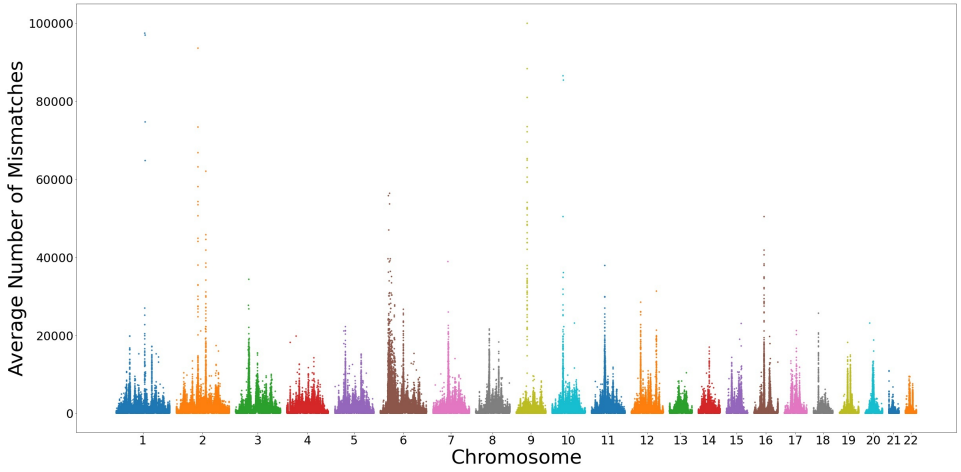


**Figure 6** Manhattan Plot for the average number of mismatches in each block at every site using all autosomal chromosomes in the UK Biobank data.

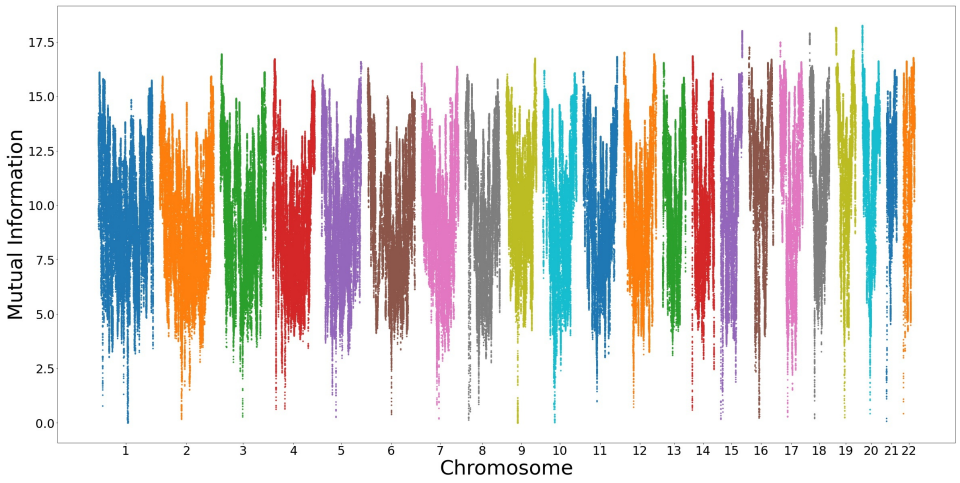


**Figure 7** Manhattan plot for the mutual information at each site using all 22 autosomal chromosomes in the UK Biobank data.

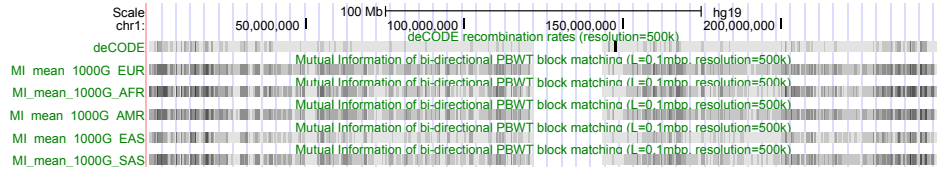


**Figure 8** Mutual information (MI) calculated by bi-directional PBWT (L = 0.1 Mbps) in 1000 Genomes Project data and deCODE recombination rate at 500 kbps scale. The MI's were calculated for each population separately.

## 3.3 Correlation between Mutual Information and Genetic Map

The mutual information was calculated for all sites in the UK Biobank data. Figure 7 shows the Manhattan plot for the mutual information at each site using all 22 autosomal chromosomes. The minimum length of the block was set to 0.5 Mbps. We also calculated the mutual information in 1000 Genomes Project data using the minimum length cut-off of 0.1 Mbps. Figure 8 shows the mutual information for each population in 1000 Genomes Project using chromosome 1 and the deCODE [7] recombination rate (cM/Mbps) at 500k resolution. We also computed the correlation between the mutual information and deCODE recombination map using the UK Biobank data and 1000 Genomes Project data in chromosome 1 (L = 0.1 Mbps). The Spearman's rank correlation coefficient for chromosome 1 of all participants was 0.85 and the European population in 1000 Genomes Project was 0.87. The correlation coefficient for the European population is slightly higher as expected when compared to the deCODE genetic map. For UK Biobank, the Spearman's rank correlation coefficient was 0.82 using all participants. In general, the mutual information correlates with the recombination rate at low resolution.

# 4 Conclusions and Discussion

The PBWT data structure provides a concise representation of a haplotype panel which enables efficient exact haplotype matching. PBWT data structures have been used for genotype imputation [8, 12], building a representation of an ancestral recombination graph

(ARG) [13], query search [9], and detecting Identical by Descent (IBD) segments [10, 15, 6]. In this work, we presented an extension of the original PBWT data structure to bi-directional. To achieve the time and memory efficiency that is required for large biobank-scale data, we designed a set of efficient algorithms. Specifically, the algorithms aim to efficiently identify blocks of matches that are present in both directions of the PBWT instances around each variant site.

Bi-directional PBWT allows the investigation of matching blocks at each site beyond the currently active site in PBWT. Therefore, it provides flexibility in detecting matches in otherwise rigid exact haplotype matches in the original PBWT. In our results, we showed that allowing only one mismatching site will have some $\sim 3 - 43\%$ increase in the reported area involved in a matching block. The presented algorithms for allowing one mismatch can be extended to multiple mismatching sites occurring with a minimum distance. Assuming that blocks of matches are less likely to occur by random, the divergence values in the forward PBWT can be updated using the block information in the reverse PBWT. As a result, approximate haplotype matches can be enumerated in the forward PBWT. Therefore, it can provide an efficient haplotype matching solution that can tolerate genotyping errors.

We also showed that the mutual information (MI) obtained by bi-directional PBWT can be used to estimate recombination rates at low resolution. The Spearman correlation coefficient between the MI of the European samples in the 1000 Genomes Project and the deCODE genetic map was 0.87. The mutual information here certainly does not provide a fine resolution map, however, it does highlight another application of the bi-directional PBWT. Another approach to determine the recombination rates can be the calculation of *edit distance* of two block sets of matches in the forward and backward PBWT, where the edit distance is defined as the minimal number of sequence reassignments to achieve similar blocks in both directions.

## References


1 Jarno Alanko, Hideo Bannai, Bastien Cazaux, Pierre Peterlongo, and Jens Stoye. Finding all maximal perfect haplotype blocks in linear time. *Algorithms for Molecular Biology*, 15(1):2, 2020.

2 Clare Bycroft, Colin Freeman, Desislava Petkova, Gavin Band, Lloyd T Elliott, Kevin Sharp, Allan Motyer, Damjan Vukcevic, Olivier Delaneau, Jared O'Connell, et al. The UK Biobank resource with deep phenotyping and genomic data. *Nature*, 562(7726):203–209, 2018.

3 1000 Genomes Project Consortium et al. A global reference for human genetic variation. *Nature*, 526(7571):68, 2015.

4 Luís Cunha, Yoan Diekmann, Luis Kowada, and Jens Stoye. Identifying maximal perfect haplotype blocks. In *Brazilian Symposium on Bioinformatics*, pages 26–37. Springer, 2018.

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

6 William A Freyman, Kimberly F McManus, Suyash S Shringarpure, Ethan M Jewett, Katarzyna Bryc, The 23, Me Research Team, and Adam Auton. Fast and Robust Identity-by-Descent Inference with the Templated Positional Burrows–Wheeler Transform. *Molecular Biology and Evolution*, 38(5):2131–2151, 2021.

7 Bjarni V Halldorsson, Gunnar Palsson, Olafur A Stefansson, Hakon Jonsson, Marteinn T Hardarson, Hannes P Eggertsson, Bjarni Gunnarsson, Asmundur Oddsson, Gisli H Halldorsson, Florian Zink, et al. Characterizing mutagenic effects of recombination through a sequence-level genetic map. *Science*, 363(6425), 2019.

8 Po-Ru Loh, Petr Danecek, Pier Francesco Palamara, Christian Fuchsberger, Yakir A Reshef, Hilary K Finucane, Sebastian Schoenherr, Lukas Forer, Shane McCarthy, Goncalo R Abecasis, et al. Reference-based phasing using the haplotype reference consortium panel. *Nature Genetics*, 48(11):1443, 2016.

**9** Ardalan Naseri, Erwin Holzhauser, Degui Zhi, and Shaojie Zhang. Efficient haplotype matching between a query and a panel for genealogical search. *Bioinformatics*, 35(14):i233–i241, 2019.

**10** Ardalan Naseri, Xiaoming Liu, Kecong Tang, Shaojie Zhang, and Degui Zhi. RaPID: ultra-fast, powerful, and accurate detection of segments identical by descent (IBD) in biobank-scale cohorts. *Genome Biology*, 20(1):1–15, 2019.

**11** Ardalan Naseri, Degui Zhi, and Shaojie Zhang. Discovery of runs-of-homozygosity diplotype clusters and their associations with diseases in UK Biobank. *medRxiv*, 2020. doi:10.1101/2020.10.26.20220004.

**12** Simone Rubinacci, Olivier Delaneau, and Jonathan Marchini. Genotype imputation using the positional burrows wheeler transform. *PLoS Genetics*, 16(11):e1009049, 2020.

**13** Vladimir Shchur, Liliia Ziganurova, and Richard Durbin. Fast and scalable genome-wide inference of local tree topologies from large number of haplotypes based on tree consistent pbwt data structure. *bioRxiv*, page 542035, 2019.

**14** Elizabeth A Thompson. Identity by Descent: Variation in Meiosis, Across Genomes, and in Populations. *Genetics*, 194(2):301–326, 2013.

**15** Ying Zhou, Sharon R Browning, and Brian L Browning. A fast and simple method for detecting identity-by-descent segments in large-scale data. *The American Journal of Human Genetics*, 106(4):426–437, 2020.