Haplotype-based Parallel PBWT for Biobank Scale Data

Kecong Tang[1], Ahsan Sanaullah[1], Degui Zhi[2], and Shaojie Zhang[1]

[1] Department of Computer Science, University of Central Florida, Orlando, FL 32826, USA
{kecong.tang, ahsan.sanaullah}@ucf.edu, shzhang@cs.ucf.edu
[2] McWilliams School of Biomedical Informatics, University of Texas Health Science Center at Houston, Houston, TX 77030, USA
degui.zhi@uth.tmc.edu

Abstract. Durbin's positional Burrows-Wheeler transform (PBWT) enables algorithms with the optimal time complexity of $O(MN)$ for reporting all vs all haplotype matches in a population panel with M haplotypes and N variant sites. However, even this efficiency may still be too slow when the number of haplotypes reaches millions. To further reduce the run time, in this paper, a parallel version of the PBWT algorithms is introduced for all versus all haplotype matching, which is called HP-PBWT (haplotype-based parallel PBWT). HP-PBWT parallelly executes the PBWT by splitting a haplotype panel into blocks of haplotypes. HP-PBWT algorithms achieve parallelization for PBWT construction, reporting all versus all L-long matches, and reporting all versus all set-maximal matches while maintaining memory efficiency. HP-PBWT has an $O((\frac{M}{T}+T)N)$ time complexity in PBWT construction, and an $O((\frac{M}{T}+T+c^*)N)$ time complexity for reporting all versus all L-long matches and reporting all versus all set-maximal matches, where T is the number of threads and c^* is the maximum number of matches (of length L or maximum divergence value for L-long matches and set-maximal matches, respectively) per haplotype per site. HP-PBWT achieves 4-fold speed-up in UK Biobank genotyping array data with 30 threads in the IO-included benchmarks. When applying HP-PBWT to a dataset of 8 million randomized haplotypes (random binary strings of equal length) in the IO-excluded benchmarks, it can achieve a 22-fold speed-up with 60 cores on the Amazon EC2 server. With further hardware optimization, HP-PBWT is expected to handle billions of haplotypes efficiently.

Keywords: PBWT · Parallel Computing · Haplotype Matching

1 Introduction

The positional Burrows–Wheeler transform (PBWT) [6] is an efficient data structure for finding haplotype matches and data compression. Construction of the PBWT can be done in linear time with respect to the number of haplotypes times the number of sites. The original PBWT paper also introduced efficient algorithms for reporting all versus all haplotype matches in a genetic panel. These algorithms have been widely applied in Identity-By-Descent (IBD) segment detection [8, 11, 20], genotype imputation [5, 13], and haplotype phasing [4, 9]. However, once the haplotype dimension of biobank panels exceeds many millions, even the PBWT may not be fast enough to satisfy the need for speed. The rapidly growing total number of genotyped individuals could reach billions in the future. This means an efficient parallel version of PBWT suitable for processing large-scale data input is in high demand.

Wertenbroek et al. [19] presented a parallelized PBWT by dividing the panel into sub panels with ranges of sites, then applied a merging algorithm to generate the final output. Their algorithm has an $O(\frac{MN}{T} + TM \log M)$ time complexity, where M is the number of haplotypes, N is the number of sites, and T is the number of threads. However, their algorithm needs to load the whole haplotype panel into memory or read the haplotype panel multiple times to perform the dividing and merging steps. Therefore, this approach is best applicable to panels with sequencing data of a relatively small sample size [1, 18].

On the other hand, the haplotype dimension of the panel is the more promising dimension to parallelize. Currently, genotyping array density datasets with millions of haplotypes [10, 16, 17] are widely used in both research and commercial applications. When compared to sequencing datasets, genotyping array density datasets have sparser sites, while typically having many more haplotypes, sometimes millions. Moreover, in the not-too-distant future, the number of genotyped samples could reach billions. Durbin's algorithms, even though enjoying a linear complexity to the number of haplotypes, need parallelization to scale up further.

In this paper, a haplotype-based parallel PBWT, HP-PBWT, is proposed, which is suitable for processing billions of haplotypes. The HP-PBWT is designed to the break dependencies in the prefix array computation and divergence array computation in Durbin's original PBWT. Furthermore, two additional efficient algorithms are designed to report all versus all L-long matches and all versus all set-maximal matches using the parallelized PBWT. HP-PBWT is memory efficient, it maintains the efficient sweeping behavior in Durbin's original PBWT and only needs an $O(M)$ memory space. HP-PBWT has $O((\frac{M}{T} + T)N)$ run time for constructing the PBWT, and $O((\frac{M}{T} + T + c^*)N)$ run time for reporting all versus all L-long matches and reporting all versus all set-maximal matches, where T is the number of threads, and c^* is the maximum number of matches per haplotype per site and in practice $c^* \ll M$.

2 Background

The description in this work follows Durbin's original work [6]. The initial input of Durbin's PBWT is X, an M by N two-dimensional binary array, which has M binary strings, x_i, which each represents a haplotype, and each haplotype has N sites. The fundamental outputs are a prefix array P_k and a divergence array D_k during the sorting process for each site $0 \leq k < N$.

The prefix array P_k stores haplotype IDs according to the colexicographic order of the haplotype prefixes of length $k+1$. P_k stores a permutation of $[0, M-1]$ such that $rev(x_{P_k[i]}[0,k])$ is lexicographically smaller than $rev(x_{P_k[i+1]}[0,k])$ for all i, where $rev(a)$ is the reverse of a string a. The term Y_k in Durbin's PBWT refers to a permutation of all sites in X at k-th location X_k, which is sorted based on the haplotype IDs in P_{k-1} such that $Y_k[i] = X_k[P_{k-1}[i]] = x_{P_{k-1}[i]}[k]$.

The divergence array D_k indicates the length of the match at k between two adjacent haplotypes in the sorted order of P_k. Therefore, if $P_k[a] = i$, $P_k[a-1] = i'$, and $D_k[i] = j$, then, $x_i[k-j+1, k] = x_{i'}[k-j+1, k]$ and $x_i[k-j] \neq x_{i'}[k-j]$. Note that the definition of the divergence array here is slightly different from Durbin's original definition. Durbin defined the divergence value as storing the starting position of the match and the values were permuted by the prefix array sorting.

A match between sequences x_a and x_b on $[i,j]$ is locally maximal if $x_a[i,j] = x_b[i,j]$ and it cannot be extended in either direction. I.E. if $x_a[i-1] \neq x_b[i-1]$, and $x_a[j+1] \neq x_b[j+1]$. Given a length cut-off L, haplotypes x_a and x_b have an L-long match on $[i,j]$, if $[i,j]$ is a locally maximal match and $j - i + 1 \geq L$. Durbin's Algorithm 3 outputs all L-long matches between all pairs of haplotypes. In this paper, this is referred to as outputting all versus all L-long matches.

A set-maximal match is defined on a haplotype x_a and a set of haplotypes, X. If x_a has a set-maximal match to x_b within X on $[i,j]$, then there is no larger match that contains it. I.E. $\forall x_c \in X$, $x_a[i-1,j] \neq x_c[i-1,j]$ and $x_a[i,j+1] \neq x_c[i,j+1]$. Durbin's Algorithm 4 outputs the set-maximal matches between x_d and $X \setminus \{x_d\}$ for all $x_d \in X$. In this paper, this is referred to as outputting all versus all set-maximal matches.

3 Methods

The focus of this work is to reduce the time complexity of the M dimension, the N dimension remains the same. First, a parallel prefix sum algorithm is used to compute the prefix array in parallel. Second, D_k is calculated by dividing Y_k into T partition blocks with T threads independently in parallel. Third, an efficient fine-grained parallel algorithm is designed to report all versus all L-long matches. At the end of this section, all versus all set-maximal matches are also reported in parallel with a similar algorithm. These algorithms have $O(\frac{M}{T}+T)$ span per site for PBWT construction and $O(\frac{M}{T}+T+c^*)$ span per site for reporting all versus all L-long matches and reporting all versus all set-maximal matches, where T is the number of threads, and c^* is the maximum number of matches per haplotype per site and in practice $c^* \ll M$.

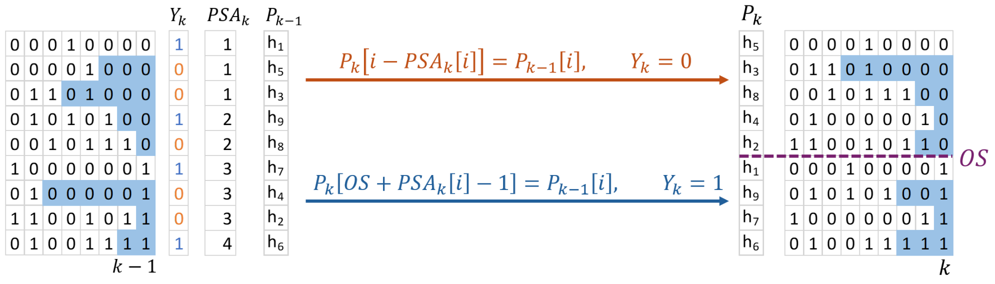
Y_k
PSA_k
P_{k-1}
P_k
$P_k[i - PSA_k[i]] = P_{k-1}[i]$, $Y_k = 0$
$P_k[OS + PSA_k[i] - 1] = P_{k-1}[i]$, $Y_k = 1$
OS
$k-1$
k

Fig. 1. Computation of new prefix array P_k from old prefix array P_{k-1} in parallel with the help of prefix sum array PSA_k. The offset is calculated by $OS = M - PSA_k[M-1]$ which is the number of zeros in Y_k.

3.1 Parallel Prefix Array Computation

Durbin's Algorithm 1 computes P_k from P_{k-1} and Y_k, by placing $i \in P_{k-1}$ into a "zero" container if $Y_k[i] = 0$ or a "one" container if $Y_k[i] = 1$ sequentially. Then, P_k is constructed by concatenating the "zero" and "one" containers. Since the process is done sequentially, this dependency has to be removed in order to execute in parallel.

The P_k construction of Durbin's PBWT Algorithm 1 is converted into a prefix sum problem. The first step is to run prefix sum on Y_k to create a PSA_k such that $PSA_k[i] = \sum_{j=0}^{i} Y_k[j]$. An important property of the PSA_k is, the PSA_k indicates the starting location of "1"s in P_k. The $PSA_k[M-1]$ is the number of "1"s in Y_k, and the offset $OS = M - PSA_k[M-1]$ is the number of "0"s in Y_k. The prefix array is defined as $P_k = PZ_k + PO_k$, where PZ_k is the "zero" container and PO_k is the "one" container. The PZ_k holds all haplotype indices i such that $Y_k[i] = 0$ by the order in P_{k-1}. Similarly, the PO_k holds the haplotype indices i such that $Y_k[i] = 1$. A mapping example is introduced from P_{k-1} to P_k according to PSA_k in Figure 1. The PO_k part of P_k can be populated by $P_k[OS + PSA_k[i] - 1] = P_{k-1}[i]$. Since PSA_k only increases by one when $Y_k[i] = 1$, the value at $PSA_k[i]$ for haplotype $P_{k-1}[i]$ is the exact one-based index of $P_{k-1}[i]$ in PO_k. The PZ_K is the first part of P_k, the new location of a haplotype $P_k[i]$ is the old index i minus the number "1"s appeared before $Y_k[i]$ which is $PSA_k[i]$. Then PZ_k part of the P_k can be computed by $P_k[i - PSA_k[i]] = P_{k-1}[i]$. There is no dependency after creating the PSA_k, so P_k can be populated in parallel.

Therefore, the computation of the PBWT prefix array has been converted to a prefix sum problem. The prefix sums of Y_k are computed in parallel. First, the input is divided into T partition blocks. Then local prefix sums within the partition blocks are calculated using one thread per partition block in parallel. Now each partition block b has its locally correct prefix sum values. To acquire the final globally correct prefix sum values, each value in partition block b has to be added a proper offset, which is the last prefix sum value of partition block $b-1$. These offsets need to be computed and added from each partition block sequentially. At the end, the corresponding offset is added to each partition block in parallel. The offset calculation is the only communication stage, and it has an $O(T)$ time complexity. This version has $O(M+T)$ work and $O(\frac{M}{T} + T)$ span per site. If T is large, then the parallel prefix sum in Section II.E of [2] (referred to as the halving merge algorithm) can be used to reduced the $O(T)$ term to $O(\log T)$ for a total work of $O(M)$ and span of $O(\frac{M}{T} + \log T)$ per site. After computing PSA_k, the haplotype IDs are mapped to the correct final prefix array location. Each haplotype takes $O(1)$ time to map and all M haplotypes are processed in parallel (See Appendix A Algorithm 1 for details). Therefore the parallelized prefix array computation algorithm as described has $O(M)$ work and $O(\frac{M}{T} + T)$ span ($O(\frac{M}{T} + \log T)$ span per site if the algorithm of [2] is used), where T is the number of threads, and T also equals to the number of partition blocks. The $O(\frac{M}{T} + T)$ span parallel prefix sum algorithm is used in this work.

3.2 Parallel Divergence Array Computation

In Durbin's algorithm, D_k is computed by sweeping through P_{k-1} keeping track of the minimum matching lengths of haplotypes that have "0" and "1" in Y_k (p and q respectively in Durbin's Algorithm 2) through

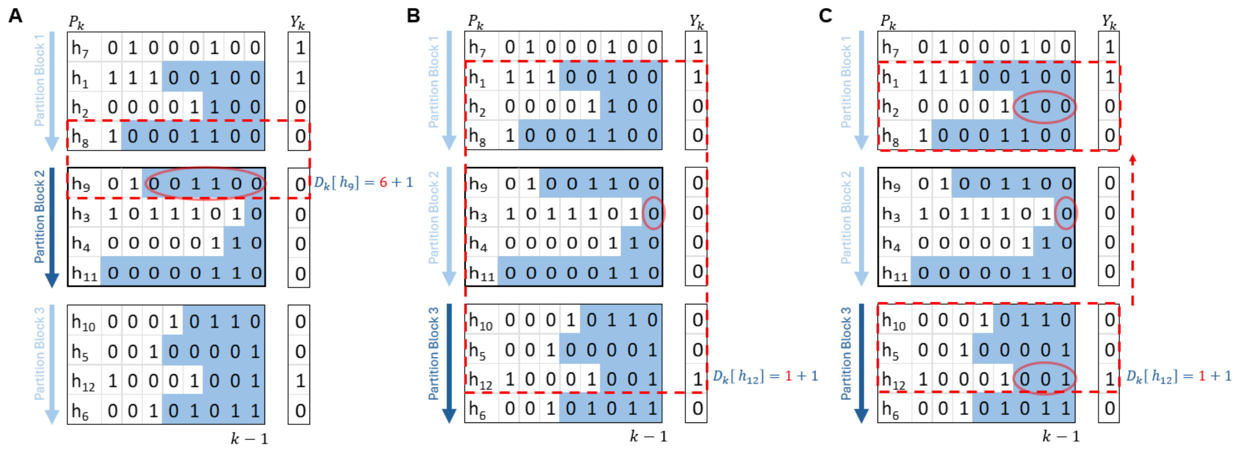

Fig. 2. Cross block upper search during divergence value computation. $D_k[h_9]$ (in A) needs a short upper search, and $D_k[h_{12}]$ (in B) needs a long upper search that cross block. Additional optimization (in C) by skipping block(s) is designed for the long upper search.

M haplotypes during the process of creating P_k. The algorithm checks if a haplotype still matches to its previous upper neighbor haplotype in P_{k-1} at Y_k site.

On the contrary, once a site is divided into partition blocks, and each block is assigned with one thread, to calculate all the divergence values within a single partition block, the passing down values (p and q in Durbin's Algorithm 2) should not be acquired from the thread that assigned to the previous partition block. Otherwise, the whole process becomes sequential. Now the challenge becomes how to acquire the correct passing down divergence values for each partition block in parallel.

Two necessary initial divergence values are calculated for each partition block by searching the previous partition block(s). To further explain, the two divergence values are: the divergence value of the first haplotype in the partition block that has "0" value in the Y_k, and the first haplotype in the partition block that has "1" value in the Y_k. Call the n-th partition block $i \in [\lceil \frac{M(n-1)}{T} \rceil, \lceil \frac{Mn}{T} \rceil)$, then, all the divergence values of the n-th partition block are $D_k[P_k[i]]$. These two divergence values can be computed by checking some upper b-th partition block(s) that $b < n$. The first divergence value is $D_k[P_k[j]]$ such that $j = Min(j \in [\lceil \frac{M(n-1)}{T} \rceil, \lceil \frac{Mn}{T} \rceil))$ and $Y_k[j] = 0$. The second divergence value is $D_k[P_k[j]]$ such that $j = Min(j \in [\lceil \frac{M(n-1)}{T} \rceil, \lceil \frac{Mn}{T} \rceil))$ and $Y_k[j] = 1$. For any of the two divergence values, $D_k[P_k[j]]$ that $Y_k[j] = v$ and $v = 0$ or 1, $D_k[P_k[j]]$ is computed by: first, find the index si that $Y_k[si] = v$, $si < j$, and $\forall i \in (si, j)$ and $Y_k[si] \neq v$; then, $D_k[P_k[j]] = Min(D_k[P_k[i]])$ that $i \in (si, j]$. For each partition block, Algorithm 2 in Appendix A searches the two divergence values in the previous partition block(s). After computing these two divergence values, it can simply loop through the partition block with Durbin's Algorithm 2 to compute the rest of the divergence values within this partition block in a single thread. The searching step is called upper search. This upper search may cross multiple partition blocks, and it does have a worst case $O(M)$ time complexity, but this is unlikely to happen since the selected markers in the biobank data panels usually do not have nearly singleton minor allele frequency. Optimizations are applied to prevent the long upper search from happening. The first optimization is verifying if the haplotype j is the first "1" in Y_k, by checking if $PSA_k[P_k[j]] = 1$, if so, this divergence value is set to 0, and the upper search isn't conducted.

The upper search examples are shown in Figure 2 which compute the two passing down divergence values for $D_k[h_9]$ in partition block 2 and $D_k[h_{12}]$ in partition block 3. Computing $D_k[h_9]$ is simply $D_k[h_9] = D_{k-1}[h_9] + 1$, since the match between haplotype h_9 and h_8 continues. Haplotype h_{12} will be sorted after haplotype h_1, so $D_k[h_{12}]$ is minimum divergence value ($D_k[h_3]$) between h_{12} and h_2. It has to reach partition block 1 to find the first upper haplotype h_1 that has the same value "1" to haplotype h_{12} at k-th site.

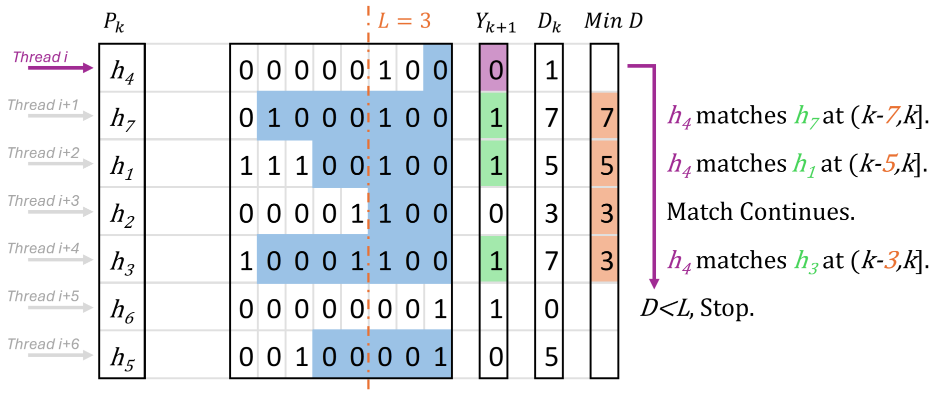

Fig. 3. Reporting all versus all L-long matches where $L = 3$ and sorted at k site in parallel. *Thread i* reports matches for haplotype h_4, it loops to haplotype h_6 since $D_k[h_6] < L$, meanwhile it compares the values of h_7, h_1, h_2 and h_3 in Y_{k+1} to h_4, then reports the match between h_4 and h_7, the match between h_4 and h_1, and the match between h_4 and h_3.

An additional optimization is applied for the situation that there is small amount of "1" in Y_k, so the upper search for the haplotype with "1" may end up $O(M)$. Demonstrated in Figure 2(C), assuming there could be many partition blocks that full of "0"s between h_{12} and h_1. For each partition block $B_i = [s, e]$ with an individual thread T_i, the first step is to identify whether this partition block has "1" or not. Here, a zero-block is defined as if $PSA_k[s-1] = PSA_k[e-1]$, that is to say there are no "1"s in this partition block. If the partition block B_i has "1", the thread T_i will skip all the zero-block(s) above it and find the nearest upper partition block $B_j = [s', e']$ that has "1" by checking the upper partition blocks according to the PSA_k as shown in Appendix A Algorithm 2. Then thread T_i runs from e' to s' to find the first p that $Y_k[p] = 1$, and compute the minimum divergence value $minD_0 = D_{k-1}[P_k[h]]$ that $\forall h \in (p, e']$. In the meantime, the minimum divergence values for each the skipped zero-blocks are still needed. The minimum divergence values for the zero-blocks are computed by the thread assigned to each partition block once a partition block is identified as zero-block. T_i waits to collect these minimum divergence values for the skipped zero-blocks if they are not yet calculated. Since it is unlikely that a site has a large run of "1"s in practice, in the implementation of HP-PBWT, this optimization is only applied on the "0"s.

The divergence value computation in HP-PBWT has $O(M + Tr)$ work and $O(\frac{M}{T} + r)$ span per site, where r is the cost of the upper search and T is the number of threads. r is reduced to $O(\frac{M}{T} + T)$ work if the same optimization was applied for both "0"s and "1"s (note $T \ll M$). Then the final work and span of divergence value computation is $O(M + T)$ and $O(\frac{M}{T} + T)$ per site respectively.

3.3 Parallel Reporting of All versus All L-long matches

Given a set of P_k, D_k, and Y_{k+1}, Durbin's Algorithm 3 outputs all matches with length $\geq L$ that end at k-th site. The algorithm has two basic steps. Step 1, acquire a precise matching block range $[s, e]$ that $\forall i \in [s, e]$ $D_k[P_k[s]] \geq L$, $D_k[P_k[e]] \geq L$, $D_k[P_k[s-1]] < L$, and $D_k[P_k[e+1]] < L$. That is to say all pairs of prefixes of haplotypes within the block have longest common suffixes of length at least length L. Step 2, output a match between every pair of haplotypes with differing values in Y_{k+1} within $[s, e]$.

In parallel execution, the simplified Algorithm 3 in Appendix A reports matches for each haplotype in an individual thread. For each haplotype $P_k[h]$, it loops through the haplotypes $P_k[i] \in (h, M)$ until $D_k[P_k[i]] < L$, in the meantime it outputs matches if $Y_{k+1}[P_k[h]] \neq Y_{k+1}[P_k[i]]$. Figure 3 provides an example of thread i reports L-long matches for haplotype h_4 independently. In detail, this algorithm does have $O(\frac{M^2}{2})$ work, since there may be $O(M^2)$ matches at a site. If c^* is used to represent the maximum number of haplotypes that match with haplotype i length L or more ending at the current site for all i, HP-PBWT has $O(\frac{M}{T} + T + c^*)$ span per site for reporting all versus all L-long matches, where T is the number of threads (note that in practice $c^* \ll M$).

3.4 Parallel Reporting All versus All set-maximal matches

A set-maximal match is a collection of the longest matches ending at k site to a single haplotype. $[s, e]$ is a range in P_k that contains all the longest matches with haplotype i ending at k. If a match in $[s, e]$ can be extended further, then there are no set-maximal matches ending at k for haplotype i. For a haplotype $P_k[i]$, the set-maximal matches at site k are reported by: First, compute $D_{Max} = Max(D_k[P_k[i]], D_k[P_k[i+1]])$, find range $[s, e]$ such that $\forall j \in [s, e], D_k[P_k[j]] = D_{Max}$, $j \neq i$, $D_k[P_k[s]] < D_{Max}$ and $D_k[P_k[e+1]] < D_{Max}$; Second, check if any $j \in [s, e], Y_{k+1}[P_k[j]] = Y_{k+1}[P_k[i]]$, if so, haplotype $P_k[i]$ does not have set-maximal matches at site k location, and stop; Third, if it passes the second step, output a set maximal match between haplotypes $P_k[i]$ and $P_k[j]$ for all $j \in [s, e]$ with a match length of D_{Max}.

The scan up and scan down to find range $[s, e]$ are light tasks since the scan up only has to reach the location j' that $D_k[P_k[j']] \neq D_{Max}$, similarly for the scan down. At this point, it is not necessary to apply complicated parallel implementation to report all versus all set-maximal matches. Instead, the parallelization is simply done by paralleling the outer loop that loops through all the haplotypes. The total work to find all the ranges $[s, e]$ is $O(c)$, where c is the sum of the number of haplotypes that match with haplotype i with length equal to the maximum match of any haplotype and i for all i ending at the current site. So, the total work among all M haplotypes is $O(M + c)$, and the span is $O(\frac{M}{T} + T + c^*)$ per site, where c^* is the maximum number of haplotypes that match with haplotype i with length equal to the maximum match of any haplotype and i for all i ending at the current site. In practice, $c^* \ll M$.

4 Results

HP-PBWT was implemented in C# and its source code and software package are available at `https://github.com/ucfcbb/HP-PBWT`. To benchmark the performance of HP-PBWT, the following tests were carried out. The first test was done to ensure correctness (See Appendix B). Then, reporting all versus all L-long matches was benchmarked using HP-PBWT, Wertenbroek et al.'s parallel PBWT, and a corrected version of Durbin's PBWT. Finally, an IO-excluded benchmark was performed for both HP-PBWT and sequential PBWT by reporting all versus all L-long matches to evaluate the scalability of the HP-PBWT-based algorithms.

4.1 Benchmark Design

Two sets of experiments were designed: IO-included, and IO-excluded. The IO-included experiments test real-world scenarios with hard drive input and output. VCF files were used as input for the IO-included experiments. To further evaluate the scalability of HP-PBWT, IO-excluded benchmark of HP-PBWT was implemented and an IO-excluded sequential PBWT was implemented as well. The idea of the IO-excluded benchmark was to remove the IO influence. This is similar to the run times presented by Wertenbroek et al. in the main paper. The IO-excluded benchmark is performed by randomly generating IO-excluded panels and then running the report all versus all L-long match algorithm without outputting matches to the hard drive. Dependencies were added to HP-PBWT to make sure computations were not optimized out by the compiler (See Appendix C for details).

Three measurements were used to evaluate HP-PBWT: run time, speed-up, and parallel efficiency. The run time of IO-included experiments included time for both reading the input and outputting matches. For the IO-excluded experiments, the run time was measured after generating the whole panel X in memory and without outputting matches. The speed-up was calculated by the sequential run time divided by the parallel run time. The parallel efficiency was calculated by speed-up divided by the number of cores.

The IO-included tests used chromosome 20 of the 1000 Genomes Project [1], and chromosome 20 of the UK Biobank [17]. To further test the scalability of these tools, randomly generated VCF files were also used with a fixed dimension of $N = 1000$ sites and M haplotypes that $M = 1000 \times 2^{\mathrm{i}}$ where $i \in [0, 15]$. The reason that 1000×2^{15} was used as the largest input was because both Durbin's and Wertenbroek et al.'s parallel PBWT use the same VCF handling library (HTSlib [3]) that can not read datasets with $M \geq 1000 \times 2^{16}$. Thus, the largest input of these experiments was $M = 1000 \times 2^{15}$. The length cut-offs were 2000 sites for the

Table 1. IO-included run time (in seconds) for real datasets. “1KG” stands for 1000 Genomes Project, “UKB” refers to UK Biobank.

PBWT version	1KG Chr.20	UKB Chr.20
Durbin’s	236	409
Wertenbroek et al.’s T=12	320	652
HP-PBWT T=10	433	227
HP-PBWT T=12	404	199
HP-PBWT T=20	389	130
HP-PBWT T=30	448	104
HP-PBWT T=40	766	133
HP-PBWT T=50	874	131
HP-PBWT T=60	989	139

1000 Genomes Project chromosome 20 (the same as in Wertenbroek et al.’s benchmark), 1600 sites for the UK Biobank chromosome 20, and 30 sites for the randomly generated files.

The IO-excluded HP-PBWT was benchmarked against the IO-excluded sequential PBWT with $M = 1000{\times}2^{i}$ haplotypes for $i \in [0, 31]$ and a fixed dimension of $N = 100$ sites. Since the M dimension was to reach 2 billion, only 100 sites were generated for the sake of memory capacity. The length cut-off was $L = 50$. To create stable run times all the IO-included tests were executed 10 times. For the IO-excluded tests, each experiment were executed 10 times for $i \in [0, 9]$, 4 times for $i \in [10, 17]$, and 2 times for $i \in [18, 31]$.

Tests were carried out for Wertenbroek et al.’s parallel PBWT to find out which number of threads (for $T \leq 60$) had the best run time, in which when $T = 12$ Wertenbroek et al.’s parallel PBWT performed the best. This is consistent with Wertenbroek et al.’s observation. All Wertenbroek et al.’s parallel PBWT’s tests in this paper use 12 threads.

The IO-included tests were executed on local servers with Intel^{R} Xeon^{R} CPU E5-2683 v4. The IO-excluded tests were executed in Windows Server 2022 on the Amazon EC2 servers which were powered by 3.6 GHz 3rd generation AMD EPYC 7R13 processors. To eliminate the interference of different programming languages and operating systems, the IO-excluded sequential PBWT was also programmed in C#. For HP-PBWT 10, 20, 30, 40, 50, and 60 cores were used in both IO-included and IO-excluded tests, meanwhile 12 cores setting was also used in the IO-included tests, since Wertenbroek et al.’s parallel PBWT had best run time on 12 cores setting.

4.2 IO-Included Benchmarks

The run times in Table 1 show Wertenbroek et al.’s parallel PBWT’s best run time (from 12 threads) did not have any speed-up comparing to Durbin’s version on either chromosome 20 of UK Biobank genotyping array data or chromosome 20 of 1000 Genomes Project sequencing data. With the same 12 thread setting, HP-PBWT had 2-fold speed-up on chromosome 20 of UK Biobank genotyping array data. Meanwhile HP-PBWT’s best run time had about 4-fold speed-up on chromosome 20 of UK Biobank genotyping array data type with 30 threads. HP-PBWT did not have any speed up in 1000 Genomes Project sequencing data, since the M dimension of 1000 Genomes Project is too small, it only has 5008 haplotypes, and the parallelization of HP-PBWT is not designed for this type of inputs. The speed-ups show that HP-PBWT can improve the run time of PBWT in large population biobank genotyping array data. Meanwhile, further improvements are needed for HP-PBWT to deal with sequencing data with less amount of haplotypes.

To test HP-PBWT’s performance on large amount of population, randomly generated VCF files were used, since there was not any real dataset with the number of samples is greater than 1 million available to the authors. Figure 4(A) shows the run times of all tools increased when the input size M increased. The speed-up in Figure 4(B) shows the Wertenbroek et al.’s parallel PBWT’s best speed-up was about 1.6-fold, HP-PBWT had a 3-fold speed-up with 12 threads and a 7.2-fold speed-up with 60 threads on the largest input comparing to the corrected Durbin’s PBWT. It shows that HP-PBWT started to gain speed-up to the corrected Durbin’s PBWT at the input of $M = 64k$, meanwhile Wertenbroek et al.’s parallel PBWT started

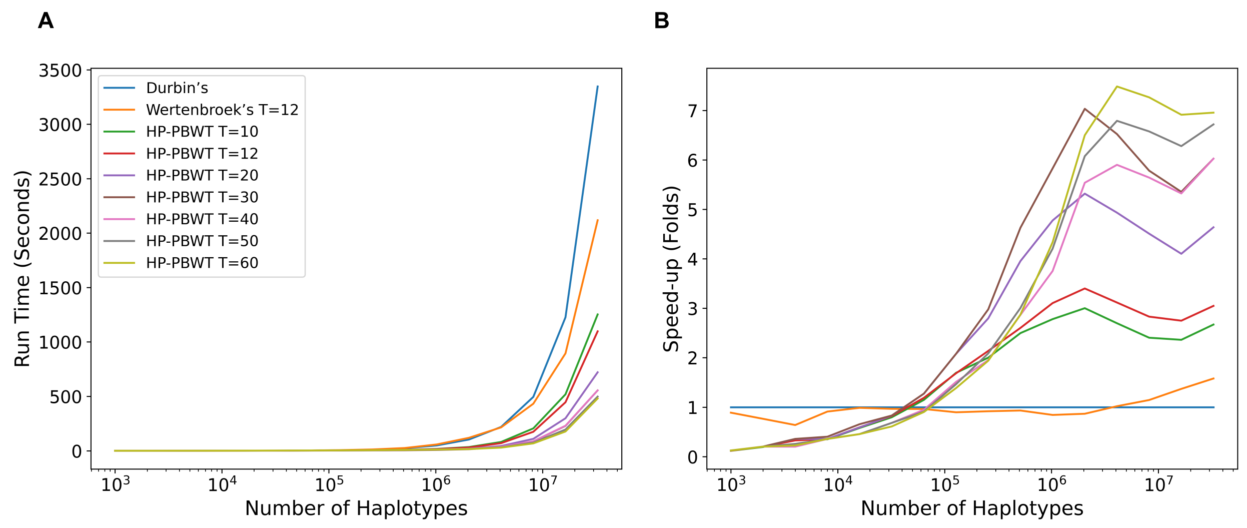

Fig. 4. IO-included run time (A) and speed-up (B) based on the random dataset across different versions of PBWT.

to gain speed-up at $M = 4m$ with 1.04-fold speed-up and reached its best speed-up of 1.6-fold speed-up at $M = 32m$.

4.3 IO-Excluded Benchmarks

Since Durbin's PBWT does not have an IO-excluded benchmark mode and Wertenbroek et al.'s benchmark mode only reads and converts hard drive files into memory. Heavy modifications should not be applied to these versions to fit the benchmark purposes. Thus, in the IO-excluded benchmarks, the IO-excluded HP-PBWT was only benchmarked with the IO-excluded sequential PBWT.

The run time in Figure 5(A) and the speed-ups in Figure 5(B) show that HP-PBWT had better performance when the size of the input increased. The best performance of HP-PBWT against the sequential PBWT was 22.2-fold at $M = 8,192,000$ with 60 threads. The benchmarks also showed the maximum number of threads (60) setting used in these experiments did not have the best run time all the time. For the largest 2 billion data input, the sequential PBWT took 5.9 hours and HP-PBWT with 50 threads execution had the shortest run time, 0.4 hours, a 13.3-fold speed-up. This benchmark was performed on the Amazon EC2 servers.

The speed-up shown in Figure 5(B) (also shown in Figure 6 in Appendix D) and parallel efficiency shown in Figure 7 in Appendix D started to fall once M increased beyond 8 million. There are a couple of potential impact factors. First, the estimated $O(\frac{M}{T}+T+c^*)$ time complexity per site for reporting all versus all L-long matches is mainly dominated by the number of haplotypes M, but once M increases to a certain level c^*, the maximum number of matches per haplotype, also becomes larger. On the other hand, this also reflects the parallel computational theory [7], that the more threads that are being used, the more idle worker time there is. Figure 8 in Appendix D shows the run time increased accordingly with the number of matches, once M reached to some millions the run time increased significantly. Second, CPU temperature also has a major impact on run time. That is why nowadays CPU and GPU cooling methods are being researched and developed constantly [12, 14, 15].

5 Discussion

In this paper, new algorithms were designed to break the dependencies that prevent Durbin's PBWT from being executed in parallel on the haplotype dimension, M. HP-PBWT algorithms enable parallel PBWT on the panel construction of prefix arrays, divergence arrays, reporting all versus all L-long matches, and reporting all versus all set-maximal matches. HP-PBWT is both memory and IO efficient. HP-PBWT can be

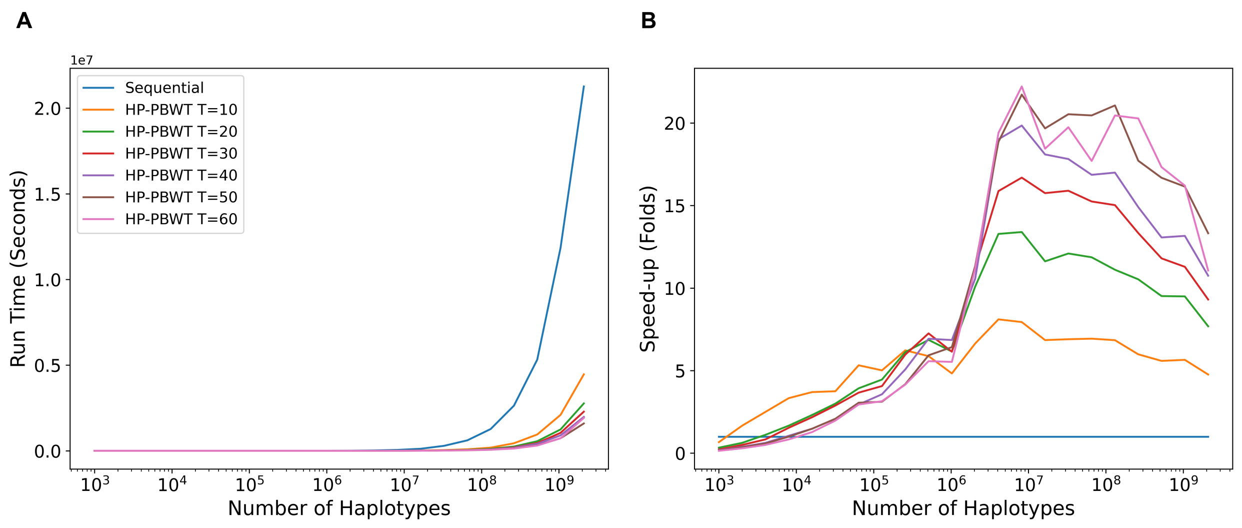

Fig. 5. IO-excluded run time (A) and speed-up (B) based on the random dataset between HP-PBWT and re-implemented IO-excluded sequential PBWT.

applied on any PBWT-based applications to leverage much larger genetic panels up to billions of haplotypes efficiently.

Currently, HP-PBWT works well on biobank scale genotyping array data, but does not have speed-up on sequencing data with small amount of population. It is likely due to the partition mechanisms in prefix array and divergence array computations, they might have too much overhead. New algorithms and optimizations can be researched for the sequencing datasets. Another possible way to speed-up the whole process is to apply both haplotype-based parallel PBWT and site-based parallel PBWT algorithms simultaneously.

Acknowledgments

This work was supported by the National Institutes of Health under award numbers R01HG010086 and R01AG081398. This research has been conducted using the UK Biobank Resource under application number 24247.

References

1. 1000 Genomes Project Consortium, et al.: A global reference for human genetic variation. Nature **526**(7571), 68 (2015)
2. Blelloch, G.E.: Scans as primitive parallel operations. IEEE Transactions on Computers **38**(11), 1526–1538 (1989)
3. Bonfield, J.K., Marshall, J., Danecek, P., Li, H., Ohan, V., Whitwham, A., Keane, T., Davies, R.M.: HTSlib: C library for reading/writing high-throughput sequencing data. GigaScience **10**(2), giab007 (2021)
4. Browning, B.L., Tian, X., Zhou, Y., Browning, S.R.: Fast two-stage phasing of large-scale sequence data. The American Journal of Human Genetics **108**(10), 1880–1890 (2021)
5. Browning, B.L., Zhou, Y., Browning, S.R.: A One-Penny Imputed Genome from Next-Generation Reference Panels. The American Journal of Human Genetics **103**(3), 338–348 (2018)
6. Durbin, R.: Efficient haplotype matching and storage using the positional Burrows–Wheeler transform (PBWT). Bioinformatics **30**(9), 1266–1272 (2014)
7. Eager, D.L., Zahorjan, J., Lazowska, E.D.: Speedup versus efficiency in parallel systems. IEEE Transactions on Computers **38**(3), 408–423 (1989)
8. Freyman, W.A., McManus, K.F., Shringarpure, S.S., Jewett, E.M., Bryc, K., 23, Team, M.R., Auton, A.: Fast and Robust Identity-by-Descent Inference with the Templated Positional Burrows–Wheeler Transform. Molecular Biology and Evolution **38**(5), 2131–2151 (2021)
9. Hofmeister, R.J., Ribeiro, D.M., Rubinacci, S., Delaneau, O.: Accurate rare variant phasing of whole-genome and whole-exome sequencing data in the UK Biobank. Nature Genetics **55**(7), 1243–1249 (2023)

10. Mathieson, I., Scally, A.: What is ancestry? PLoS Genetics **16**(3), e1008624 (2020)
11. 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**(1), 1–15 (2019)
12. Ramakrishnan, B., Alissa, H., Manousakis, I., Lankston, R., Bianchini, R., Kim, W., Baca, R., Misra, P.A., Goiri, I., Jalili, M., et al.: CPU Overclocking: A Performance Assessment of Air, Cold Plates, and Two-Phase Immersion Cooling. IEEE Transactions on Components, Packaging and Manufacturing Technology **11**(10), 1703–1715 (2021)
13. Rubinacci, S., Delaneau, O., Marchini, J.: Genotype imputation using the Positional Burrows Wheeler Transform. PLoS Genetics **16**(11), e1009049 (2020)
14. Shahi, P., Mathew, A., Saini, S., Bansode, P., Kasukurthy, R., Agonafer, D., et al.: Assessment of Reliability Enhancement in High-Power CPUs and GPUs Using Dynamic Direct-to-Chip Liquid Cooling. Journal of Enhanced Heat Transfer **29**(8) (2022)
15. Siricharoenpanich, A., Wiriyasart, S., Naphon, P.: Study on the thermal dissipation performance of GPU cooling system with nanofluid as coolant. Case Studies in Thermal Engineering **25**, 100904 (2021)
16. Stoeklé, H.C., Mamzer-Bruneel, M.F., Vogt, G., Hervé, C.: 23andMe: a new two-sided data-banking market model. BMC Medical Ethics **17**, 1–11 (2016)
17. Sudlow, C., Gallacher, J., Allen, N., Beral, V., Burton, P., Danesh, J., Downey, P., Elliott, P., Green, J., Landray, M., et al.: UK Biobank: An Open Access Resource for Identifying the Causes of a Wide Range of Complex Diseases of Middle and Old Age. PLoS Medicine **12**(3), e1001779 (2015)
18. Taliun, D., Harris, D.N., Kessler, M.D., Carlson, J., Szpiech, Z.A., Torres, R., Taliun, S.A.G., Corvelo, A., Gogarten, S.M., Kang, H.M., et al.: Sequencing of 53,831 diverse genomes from the NHLBI TOPMed Program. Nature **590**(7845), 290–299 (2021)
19. Wertenbroek, R., Xenarios, I., Thoma, Y., Delaneau, O.: Exploiting parallelization in positional Burrows-Wheeler transform (PBWT) algorithms for efficient haplotype matching and compression. Bioinformatics Advances **3**(1), vbad021 (2023)
20. 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**(4), 426–437 (2020)

Appendices

A Algorithms

Algorithm 1 Computate of prefix array P_k in parallel

```
$OS \leftarrow M - PSA[M-1]$                                   ▷ 1s' Offset, number of "0"s.
for $i \leftarrow 0$ to $M$ do                                  ▷ Parallel execute
    if $Y_k[i] = 1$ then
        $P_k[OS + PSA_k[i] - 1] \leftarrow P_{k-1}[i]$
    else
        $P_k[i - PSA_k[i]] \leftarrow P_{k-1}[i]$
    end if
end for

return $P_k$
```

Algorithm 2 Compute divergence array D_k in parallel

```
$BlockSize \leftarrow \lceil \frac{M}{T} \rceil$
for $i \leftarrow 0$ to $T$ do                                   ▷ Parallel execute
    $s \leftarrow \lceil \frac{Mi}{T} \rceil$                                   ▷ Start index of the block
    $e \leftarrow \lceil \frac{M(i+1)}{T} \rceil$                               ▷ End index of the block
    if $Y_k[s-1]$=0 then      ▷ Compute the Upper Min divergence value for haplotype with zeros
        $MinD_1 \leftarrow Max$
        if $PSA_k[s-1] = PSA_k[e-1]$ then                       ▷ No 1 appeared in this block
            for $j \leftarrow s$ to $e$ do                              ▷ Compute minimum value
                $RangeMins[i] \leftarrow \min(RangeMins[i], D_{k-1}[j])$
            end for
        end if

        while $seekBlockID \geq 0$ do                  ▷ Jump to the first upper block has 1
            $sSeek \leftarrow \lceil \frac{M(i-1)}{T} \rceil$                    ▷ Start index of the search block
            $eSeek \leftarrow \lceil \frac{M(i)}{T} \rceil$                       ▷ End index of the search block
            if $PSA_k[sSeek-1] \neq PSA_k[eSeek-1]$ then               ▷ 1 appeared in this block
                Break
            end if
            $seekBlockID --$
        end while
        $seekIndex \leftarrow (seekBlockID + 1) \times BlockSize - 1$
        $MinD_0 \leftarrow D_{k-1}[seekIndex]$
        while $seekIndex \geq 0$ and $Y_k[seekIndex] = 0$ do
            $MinD_0 \leftarrow \min(MinD_0, D_{k-1}[seekIndex])$
            $seekIndex --$
        end while
        for $b \leftarrow seekBlockID + 1$ to $i$ do     ▷ Apply the minimum values in the skip ranges
            $MinD_0 \leftarrow \min(MinD_0, RangeMins[b])$
        end for
    else                       ▷ Compute the Upper Min divergence value for haplotype with ones
        $MinD_0 \leftarrow Max$
        $seekIndex \leftarrow s - 1$
        $MinD_1 \leftarrow D_{k-1}[seekIndex]$
        while $seekIndex \geq 0$ and $Y_k[seekIndex]! = 0$ do
            $MinD_1 \leftarrow \min(MinD_1, D_{k-1}[seekIndex])$
            $seekIndex --$
        end while
    end if
    for $j \leftarrow s$ to $e$ do                ▷ Compute divergence values with Durbin's Algorithm
        $hID \leftarrow P_k[j]$                                                  ▷ haplotype ID
        if $Y_k[j] = 0$ then
            $D_k[hID] \leftarrow \min(MinD_1, D_{k-1}[hID]) + 1$
            $MinD_1 \leftarrow Max$
            $MinD_0 \leftarrow \min(MinD_0, D_{k-1}[hID])$
        else
            $D_k[hID] \leftarrow \min(MinD_0, D_{k-1}[hID]) + 1$
            $MinD_0 \leftarrow Max$
            $MinD_1 \leftarrow \min(MinD_1, D_{k-1}[hID])$
        end if
    end for
end for
```

Algorithm 3 Report All versus All L-long matches ended at k site in parallel

```
for $h \leftarrow 0$ to $M$ do                                   ▷ Parallel execute
    for $i \leftarrow h+1$ to $M$ do
        if $D_k[P_k[i]] < L$ then
            Break
        else
            if $Y_{k+1}[P_k[h]] \neq Y_{k+1}[P_k[i]]$ then
                $P_k[h]$ matches $P_k[i]$ at $(k - D_k[P_k[i]], k]$
            end if
        end if
    end for
end for
```

B Correctness

To verify the correctness of HP-PBWT, first, naive algorithms that find all versus all L-long matches and all versus all set-maximal matches ended at each site were implemented. Then the matches from the IO-excluded HP-PBWT, the IO-excluded sequential PBWT, and the naive algorithms were compared to ensure the correctness. Second, HP-PBWT's output was compared with Durbin's PBWT, and Wertenbroek et al.'s parallel PBWT on UK Biobank chromosome 20 with $L = 1600$. Wertenbroek et al.'s parallel PBWT outputted 10,348,502 matches, Durbin's PBWT outputted 10,362,502 matches, and HP-PBWT outputted 10,521,839 matches. Every match outputted by Wertenbroek et al's parallel PBWT. was outputted by Durbin's PBWT and every match outputted by Durbin's PBWT was outputted by HP-PBWT. The matches outputted by HP-PBWT were verified, they were all versus all L-Long matches. Two minor issues were found in Durbin's implementation. The first issue is it does not report matches that match to the last site. The second issue is the reporting code block can not be triggered when it reaches the last haplotype in P_k that $D_k[P_k.Last] \geq L$. After correcting these two issues, HP-PBWT and the corrected Durbin's version produce identical outputs. Furthermore, it was observed that the corrected Durbin's PBWT had nearly equal run time performance to the original Durbin's PBWT. Thus, the Durbin's run time presented in this paper are from the corrected Durbin's PBWT.

C Dependencies

Dependencies were added to IO-excluded HP-PBWT and the IO-excluded sequential PBWT to make sure none of the code blocks was skipped to create correct run times. The dependencies were created by maintaining a check sum for each haplotype of the haplotypes it matched to and the lengths of the matches. At the end one of the check sums was randomly selected and outputted.

D Additional Figures

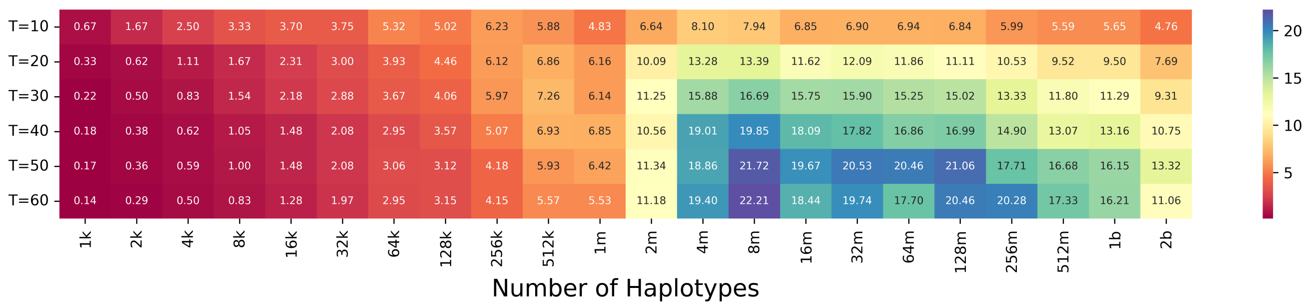

Fig. 6. Speed-up from IO-excluded tests.

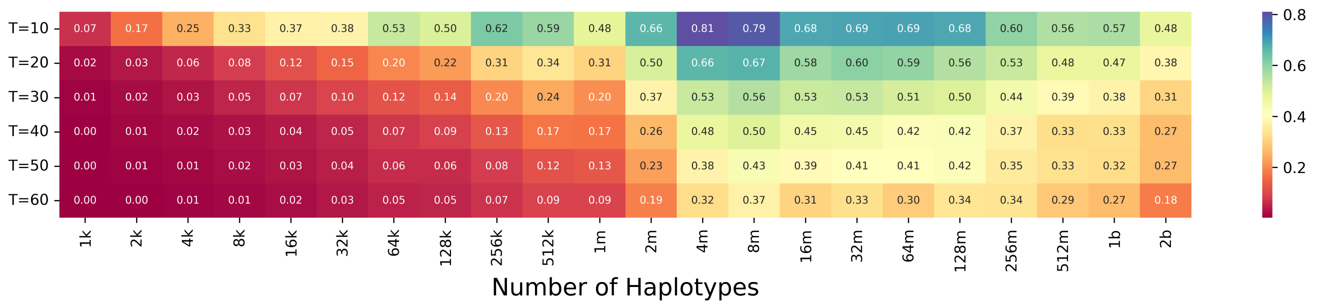

Fig. 7. Parallel efficiency from IO-excluded tests. The parallel efficiency is calculated by speed-up divided by the number of threads.

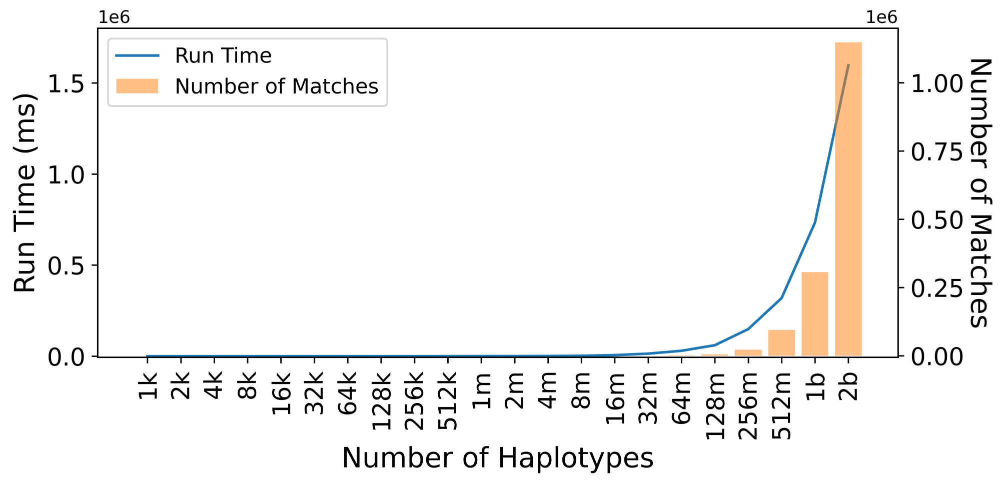

Fig. 8. The correlation between run time (in milliseconds) and number of matches, the data was collected from IO-excluded tests with 50 threads.