Genetics and population analysis

Exploiting parallelization in positional Burrows-Wheeler transform (PBWT) algorithms for efficient haplotype matching and compression

Rick Wertenbroek [1,2,*], Ioannis Xenarios [2], Yann Thoma [1,†], and Olivier Delaneau [2,†,*]

[1] School of Engineering and Management Vaud (HEIG-VD), HES-SO University of Applied Sciences and Arts Western Switzerland, Yverdon-les-Bains, 1401, Switzerland and
[2] University of Lausanne, Lausanne, 1015, Switzerland.
[†] Co-supervised this work.

[*] To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Summary: The positional Burrows-Wheeler transform (PBWT) data structure allows for efficient haplotype data matching and compression. Its performance makes it a powerful tool for bioinformatics. However, existing algorithms do not exploit parallelism due to inner dependencies. We introduce a new method to break the dependencies and show how to fully exploit modern multi-core processors.
Availability: Source code and applications are available at https://github.com/rwk-unil/parallel_pbwt
Contact: rick.wertenbroek@unil.ch, olivier.delaneau@unil.ch
Supplementary information: Supplementary data are available at *Bioinformatics Advances* online.

1 Introduction

The positional Burrows-Wheeler transform (PBWT) data structure allows for the development of efficient matching algorithms between haplotypes (Durbin, 2014). That is, the PBWT allows matching in linear time relative to the number of haplotypes instead of the quadratic time of a naive all-vs-all approach. Another advantage is the impressive data compression rate made possible by this data structure. This makes the PBWT, and associated algorithms, a core component of bioinformatics tools, such as *Beagle, EagleImp, GLIMPSE* or *XSI* (Browning et al., 2021; Wienbrandt and Ellinghaus, 2022; Rubinacci et al., 2021; Wertenbroek et al., 2022), and bioinformatics courses (Gagie et al., 2022). However, algorithms relying on the PBWT for processing haplotypes over a genomic region exhibit a positional dependency, i.e., the state of the structure at a given position (genomic locus) depends on the previous position. This makes it hard to parallelize. Others (Sanaullah et al., 2021; Wang et al., 2022; Shakya et al., 2022) have proposed methods to make PBWT algorithms more efficient but are still sequential. In this paper we introduce a method to manage the dependency and split the problem for parallel execution. We show that with this new method haplotype matching algorithms can achieve a speed-up of up to 8× on a modern 12-core processor.

2 Methods

The algorithms presented in (Durbin, 2014) for haplotype matching and compression rely on two key internal data structures: the positional prefix array a_k which represents the ordering of haplotypes at position k and the divergence array d_k which stores the position where a haplotype differs from the previous one in the current ordering[1]. The a_k and d_k arrays for a position k are built from the arrays of the previous position a_{k-1} and d_{k-1}. This dependency propagates back until the initial arrays a_0 and d_0 which are given. The positional prefix array a_0 represents the arbitrary order the haplotypes come in from the input data, at each position the haplotypes are reordered given the genotype they carry at that position, either a 0 (reference genotype) or a 1 (alternative genotype). So when generating a_1 all haplotypes with the reference genotype (0) at the first position ($k = 0$) come before those that have an alternate genotype (1) at position $k = 0$. This is equivalent to a *radix sort* (also known as *digital sort*

[1] For definitions see (Durbin, 2014) the same nomenclature is used here.

© The Author(s) 2023. Published by Oxford University Press.
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted reuse, distribution, and reproduction in any medium, provided the original work is properly cited.

Downloaded from https://academic.oup.com/bioinformaticsadvances/advance-article/doi/10.1093/bioadv/vbad021/7067758 by Fachhochschulen user on 10 March 2023

or *bucket sort*). Therefore, at each position k, the k previous genotypes (reverse prefix) dictate the position of that haplotype. The divergence array d_k is generated at the same time by keeping track of at which position k, previously matching haplotypes stop matching (don't share the same genotype anymore). These two arrays are key in PBWT-based algorithms for matching or compression. To start processing at an arbitrary position k the arrays a_k and d_k must be known (iteratively computed from the starting position 0 up to k). This dependency makes it difficult to split a genomic region with N loci, $k \in [0, N[$ between separate threads for parallel processing.

2.1 Splitting the genomic region and breaking the dependency

A key observation is that if we generate a_k and d_k from a previous position $k - b$ (over a chunk of b previous loci, $b < k$) with initial arrays a_0 and d'_{k-b} (an array filled with the value $k-b$) instead of the actual arrays a_{k-b} and d_{k-b}, then for all haplotypes, except the ones that are identical over $k - b$ to k, the computed values in the a_k and d_k arrays will be correct. That is, as if computed from the start, i.e., starting at $k = 0$ with a_0 and d_0 and iterating over k loci instead of only b loci. This means we have an approximated version of a_k and d_k that can only have wrong values for groups of identical haplotypes over the chunk of b loci. For a because they cannot be ordered given the b observed loci and for d because they differ at a loci before $k - b$. Because d'_{k-b} was initialized with $k - b$ the condition $d_k[i] = k - b$ lets us know for which indices i the values of a_k and d_k might be wrong. **Keypoint:** The correction of arrays a_k and d_k computed from position $k - b$ (b steps) with a_0 and d'_{k-b} instead of a_{k-b} and d_{k-b} can be done in a single step if the correct a_{k-b} and d_{k-b} are known. **Strategy:** It is possible split the genomic region of N loci into t chunks of $b = N/t$ loci for t threads to handle in parallel. Each thread can compute the approximated a_k and d_k arrays (end of the chunk) from the position $k - b$ (start of the chunk), with a_0 and d'_{k-b} instead of the actual arrays for the start of each chunk. The a_k and d_k arrays at the end of the first chunk will be correct because the first chunk is supposed to start with a_0 and d_0 (note, $d'_0 = d_0$). Then, we can use the *keypoint* above to correct the remaining $t - 1$ a and d arrays in $t - 1$ sequential steps. Because t will typically be small (e.g., 2-64 threads), the number of steps executed sequentially is small compared to the total number of steps N (e.g., in the millions). Also, the bigger the chunk the smaller the chance to have identical haplotypes, the less time will be required to correct the a and d arrays. Once the t arrays are generated the heavier algorithms (matching, compression, etc.) can be launched in parallel with t threads. The process is illustrated in the supplementary materials figures SP1-SP4.

2.2 Algorithms to correct approximated a and d arrays

The method to correct a and d is decomposed into two algorithms; **Algorithm 1** shows how to fix a_k and d_k, between a $start$ and $stop$ index, given the arrays at $k - b$. The start and stop indices represent a group of identical haplotypes over loci $k - b$ to k. To rearrange the haplotypes in a_k between $start$ and $stop$, they require to follow the order given in a_{k-b}. To do so, the positions of the haplotypes in the previous chunk are looked up in a^{-1}_{k-b}, which is the inverse of the positional prefix array a_{k-b} (see Algorithm 2). These positions are then sorted in incremental order and finally the correct order is set in a_k by referring to the haplotype number in a_{k-b} given the incrementally sorted indices.

Now that the group of identical haplotypes are ordered correctly (correct values in a_k), we need to fix the divergence values, the first value at position $start$ will already be correct because it refers to the previous non matching haplotype. For the other values they now need to be updated to reflect the divergences in the previous chunk given the corrected ordering. Although the haplotypes are grouped together in the current chunk, they might have other haplotypes in between them in the previous chunk. This requires to scan for the biggest value of d between the previous haplotype and the current one referring to the ordering in the previous chunk, similarly to what is done with p, q of algorithm 2 presented in (Durbin, 2014).

Algorithm 1: Correction of positional prefix and divergence arrays a_k, d_k between $start$ and $stop$ given previous arrays a_{k-b}, d_{k-b}

```
Initialization, create array $arr[\mathrm{stop} - \mathrm{start}]$
for $i := start\ to\ stop - 1$ do
    $arr[i - start] := a^{-1}_{k-b}[a_k[i]]$
end
sort $(arr)$
for $i := start\ to\ stop - 1$ do
    $a_k[i] := a_{k-b}[arr[i - start]]$
end
for $i := start + 1\ to\ stop - 1$ do
    scan_start $:= arr[i - start - 1] + 1$
    scan_stop $:= arr[i - start] + 1$
    $d_k[i] =$ max_element$(d_{k-b}$[scan_start : scan_stop]$)$
end
```

Algorithm 2 allows to determine groups of matching haplotypes over $k - b$ to k and calls algorithm 1 to fix the values in a_k and d_k. To do so algorithm 2 has to generate the array a^{-1}_{k-b} which is required for algorithm 1. This is done by looping through all entries of a_{k-b} which are haplotype identifiers, so if $a_{k-b}[0] = id_x$ it means the first haplotype is id_x, therefore $a^{-1}_{k-b}[id_x] = 0$. So a maps positions to identifiers and a^{-1} maps identifiers to positions. Then algorithm 2 iterates over the d array keeping track of haplotype groups matching over $k - b$ to k, for each of such groups it will call algorithm 1 to correct the a and d arrays.

Algorithm 2 is used to sequentially correct the approximate a and d arrays generated in parallel with the strategy proposed in 2.1 (see algorithm P in section 2.3). A step-by-step example of the execution of algorithm 1 and 2 is provided in the supplementary materials and illustrated with figures S1, S2.

Algorithm 2: Correction of positional prefix and divergence arrays a_k, d_k given correct previous arrays a_{k-b}, d_{k-b}

```
initialization, create array $a^{-1}_{k-b}[M]$, group_index := 0
// Fill $a^{-1}_{k-b}$ array, reciprocal of $a_{k-b}$ array
for $i := 0\ to\ M - 1$ do
    $a^{-1}_{k-b}[a_{k-b}[i]] := i$
end
// Iterate over the divergence array to find and fix matching groups
for $i := 0\ to\ M - 1$ do
    if $d_k[i] \neq p$ then
        if $i - group\_index > 1$ then
            Algorithm 1 with start := group_index and stop := $i$
        end
        group_index := $i$
    end
end
if $M - group\_index > 1$ then
    Algorithm 1 with start := group_index and stop := $M$
end
```

2.3 From sequential to parallel algorithm

The sequential PBWT-based algorithms can be summarized with algorithm S which loops over all N genotype loci and alternates between

Downloaded from https://academic.oup.com/bioinformaticsadvances/advance-article/doi/10.1093/bioadv/vbad021/7067758 by Fachhochschulen user on 10 March 2023

updating the a and d arrays and running the matching algorithm or compression step.

```
Algorithm S: Sequential PBWT-based algorithm
  Constants: $N$: #genotype loci, $M$: #haplotypes
  Initialization, create array $a_0 = [0, 1, 2, ..., M-1]$ and
  $d_0 = [0, ..., 0]$
  for $k := 0$ to $N$ do
    Run matching (e.g., algorithm 3 or 4 from (Durbin, 2014)) or
    compression step
    Generate $a_{k+1}$ and $d_{k+1}$ from $a_k$ and $d_k$
  end
```

Our parallel implementation is described in algorithm P. The algorithm relies on the *keypoint* and *strategy* presented above. The algorithm starts by a parallel step to generate the approximate a and d arrays, then runs a small sequential loop to correct these arrays with algorithm 2 presented above. Finally, it launches T threads that will each run algorithm S with the heavier matching or compression algorithms. Each thread handles a chunk of the genomic region starting at positions $k \in \{0, b, 2b, 3b, ..., (T-1) \cdot b\}$ with the now available and correct a_k, d_k arrays (instead of a single thread running algorithm S over the whole genomic region starting at $k = 0$ with a_0 and d_0 and finishing at N).

```
Algorithm P: Parallel PBWT-based algorithm
  Constants: $N$: #genotype loci, $M$: #haplotypes, $T$: #threads
  $b = N/T$ // Chunk size
  // A: Parallel generation of approximate a and d arrays
  Launch $T-1$ threads with algorithm 2 from (Durbin, 2014) for b
  steps starting from positions $k \in \{0, b, 2b, 3b, ..., (T-2) * b\}$
  with arbitrary $a_k = [0, 1, 2, ..., M-1]$ and $d_k = [k, ..., k]$
  that will generate the approximate $a_i$ and $d_i$ arrays (with
  $i \in \{b, 2b, 3b, ..., (T-1) * b\}$)
  // B: Sequential correction of the approximate arrays $a_i, d_i$
  for $t := 1$ to $T-1$ do
    join(thread $t$)
    if $t > 1$ // (Note : $a_b, d_b$ are already correct) then
      Algorithm 2: correct $a_{t \cdot b}$ and $d_{t \cdot b}$ with $a_{(t-1) \cdot b}$ and
      $d_{(t-1) \cdot b}$
    end
  end
  // C: Parallel run of matching algorithm 3 or 4
  Launch T threads running Algorithm S with e.g., algorithm 3 or 4
  from (Durbin, 2014) for b steps starting from positions
  $k \in \{0, b, 2b, 3b, ..., (T-1) * b\}$ with the now available and
  correct $a_k, d_k$ arrays
```

2.4 Time and space complexity analysis

The worst case time complexity of algorithm 1 relative to M input haplotypes (if all haplotypes match from $k-b$ to k, with a and d that require to be corrected) is quasilinear, $\mathcal{O}(M \log M)$. Algorithm 1 can be split in four steps, three loops and one sorting algorithm: The first two loops iterate over the matching haplotypes so they are $\mathcal{O}(M)$. The sort is $\mathcal{O}(M \log M)$ because it is implemented with a merge sort (Knuth, 1975). The last loop may look like it could have quadratic complexity because of the inner *max element* look-up, but it has not. The number of look-ups for the combined *max elements* is bounded by the number of haplotypes, because the array *arr* is comprised of sorted positions we have $0 \leq \text{scan\_start} < \text{scan\_stop} \leq M$ and $\sum_i (\text{scan\_stop} - \text{scan\_start})$ is bounded by M, therefore the number of look-ups in this loop is $\mathcal{O}(M)$.

The worst case time complexity of algorithm 2 relative to M input haplotypes is quasilinear $\mathcal{O}(M \log M)$. Algorithm 2 has two main parts: First, it generates a^{-1}_{k-b} which is done in M steps, therefore it is $\mathcal{O}(M)$. Second, it has to apply algorithm 1 to a number of matching haplotype groups. The time complexity of algorithm 1 is dominated by the sorting step which is $\mathcal{O}(M \log M)$. The worst case time complexity of the second part of algorithm 2 is also $\mathcal{O}(M \log M)$, because either all haplotypes match and we have a single group of size M to sort (apply algorithm 1), or we have a given number of smaller groups to sort. Because the sum of the group sizes is bounded by M, running a number of smaller sorts will require a lower or equal asymptotic number of steps than sorting all the haplotypes (e.g., apply algorithm 1 to M groups of size 1). Therefore, the worst case time complexity of running algorithm 2 is $\mathcal{O}(M \log M)$.

The space complexity is $\mathcal{O}(M)$ because a constant number of arrays of size M is required and the merge sort also has linear space complexity (Knuth, 1975).

The PBWT algorithms for matching and compression have a worst case time complexity of $\mathcal{O}(NM)$ where N is the number of genotype loci and M the number of haplotypes. Our parallel version will have a worst case time complexity of $\mathcal{O}(NM/T + TM \log M)$ where T is the number of threads. The added $\mathcal{O}(TM \log M)$ quasilinear complexity is negligible compared to the gains we have by dividing NM by T. For example, with data from (1000 Genomes Project Consortium and others, 2015): $N \geq 88,000,000$, $M = 5,008$, and T will typically be small (e.g., 2-64 threads).

3 Results

We applied the strategy above on two haplotype matching algorithms from (Durbin, 2014) and implemented them as the parallel implementation shown in algorithm P: Algorithm 3 which reports all matches between haplotypes above a given length and Algorithm 4 which reports all *set-maximal* matches between haplotypes. Fig. 1 shows the runtime of the original single-threaded algorithm (algorithm S) and its multi-threaded counterparts (algorithm P) for different number of threads t on data from (1000 Genomes Project Consortium and others, 2015). With an AMD 3900X processor. The multi-threaded implementations achieve a speed-up of 7× and 8.37× on algorithms 3 and 4 respectively running with 12 threads. Results on the Human Reference Consortium data (McCarthy et al., 2016) are available in the supplementary materials (with a similar

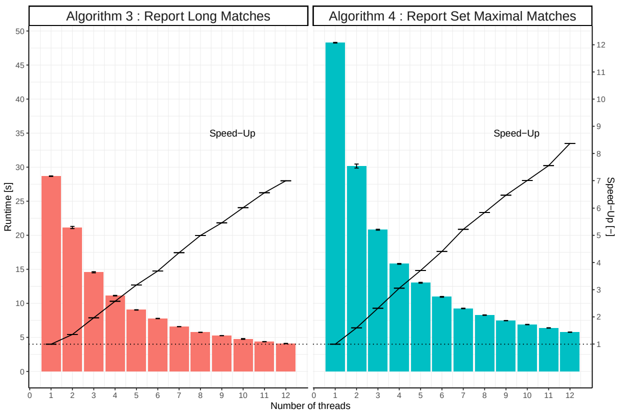

Fig. 1. Runtime and speed-up of algorithms 3 and 4 from (Durbin, 2014) and their parallel implementations running on a 12-core AMD 3900X processor (each run 10 times). Algorithm 3 reports all matches longer than 2000 genotypes (loci) between all haplotypes, Algorithm 4 reports the set maximal matches between all haplotypes. Data is chromosome 20 from (1000 Genomes Project Consortium and others, 2015) 5,008 haplotypes, 1,822,268 genotype loci.

Downloaded from https://academic.oup.com/bioinformaticsadvances/advance-article/doi/10.1093/bioadv/vbad021/7067758 by Fachhochschulen user on 10 March 2023

speed-up of $7.04\times$ and $8.16\times$ for 12 threads), as well as an example application that implements the matching algorithms and reports the results to a file to allow a direct comparison to the original software from (Durbin, 2014). The supplementary materials also provide a comparison between generating the a and d arrays sequentially with algorithm 2 from (Durbin, 2014) against our parallel implementation with sequential correction (algorithm 1 and 2) presented here (sections A and B of algorithm P) for different number of threads. Fig. SP5 shows that the parallel version followed by the sequential correction can provide a speed-up of up to $10.94\times$ when generating the a and d arrays.

4 Discussion

In this paper we have presented a method and two algorithms that allow parallel execution of PBWT-based haplotype matching algorithms. The method allows to exploit modern multi-core processors and has shown a $7\times$-$8.37\times$ reduction in execution time with 12 threads compared to the single-threaded version. For PBWT-based compression, some methods break the per loci dependency by design for better random access performance, e.g., (Wertenbroek et al., 2022). Therefore, these algorithms can be multi-threaded directly. However, compression methods that do not break this dependency, e.g., (Durbin, 2014; Li, 2016; Deorowicz and Danek, 2019; LeFaive et al., 2021) could be accelerated by the presented methods. Beside these results, the a and d arrays could be saved to a file so that subsequent runs of the algorithms could be launched in parallel directly and now these arrays can be generated efficiently in parallel thanks to Algorithm 1 and 2 presented here, with up to a $10.94\times$ reduction in time. Algorithm 2 also exhibits another interesting property: it provides the indices of groups of haplotypes that are identical over a large genomic chunk. This information could be leveraged to speed up PBWT based methods, e.g., (Hofmeister et al., 2022), by treating the whole group as a single haplotype block and avoid redundant computations.

Acknowledgements

The authors would like to thank Dr. Simone Rubinacci for his insights.

Funding

This work has been supported by the School of Engineering and Management Vaud (HEIG-VD). Olivier Delaneau is supported by SNF grant number : SNSF-PP00P3_176977

References

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

Browning, B. L., Tian, X., Zhou, Y., and Browning, S. R. (2021). Fast two-stage phasing of large-scale sequence data. *The American Journal of Human Genetics*, 108(10):1880–1890.

Deorowicz, S. and Danek, A. (2019). GTShark: genotype compression in large projects. *Bioinformatics*, 35(22):4791–4793.

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

Gagie, T., Manzini, G., and Sciortino, M. (2022). Teaching the burrows-wheeler transform via the positional burrows-wheeler transform. *arXiv preprint arXiv:2208.09840*.

Hofmeister, R. J., Ribeiro, D. M., Rubinacci, S., and Delaneau, O. (2022). Accurate rare variant phasing of whole-genome and whole-exome sequencing data in the UK Biobank. *bioRxiv*.

Knuth, D. E. (1975). Sorting by Merging. In *The art of computer programming: sorting and searching*, pages 158–168.

LeFaive, J., Smith, A. V., Kang, H. M., and Abecasis, G. (2021). Sparse allele vectors and the savvy software suite. *Bioinformatics*, 37(22):4248–4250.

Li, H. (2016). BGT: efficient and flexible genotype query across many samples. *Bioinformatics*, 32(4):590–592.

McCarthy, S., Das, S., Kretzschmar, W., Delaneau, O., Wood, A. R., Teumer, A., Kang, H. M., Fuchsberger, C., Danecek, P., Sharp, K., et al. (2016). A reference panel of 64,976 haplotypes for genotype imputation. *Nature genetics*, 48(10):1279.

Rubinacci, S., Ribeiro, D. M., Hofmeister, R. J., and Delaneau, O. (2021). Efficient phasing and imputation of low-coverage sequencing data using large reference panels. *Nature Genetics*, 53(1):120–126.

Sanaullah, A., Zhi, D., and Zhang, S. (2021). d-PBWT: dynamic positional Burrows–Wheeler transform. *Bioinformatics*, 37(16):2390–2397.

Shakya, P., Naseri, A., Zhi, D., and Zhang, S. (2022). mcPBWT: Space-efficient Multi-column PBWT Scanning Algorithm for Composite Haplotype Matching. In *Computational Advances in Bio and Medical Sciences: 11th International Conference, ICCABS 2021, Virtual Event, December 16–18, 2021, Revised Selected Papers*, pages 115–130. Springer.

Wang, V., Naseri, A., Zhang, S., and Zhi, D. (2022). Syllable-PBWT for space-efficient haplotype long-match query. *bioRxiv*.

Wertenbroek, R., Rubinacci, S., Xenarios, I., Thoma, Y., and Delaneau, O. (2022). XSI—a genotype compression tool for compressive genomics in large biobanks. *Bioinformatics*, 38(15):3778–3784.

Wienbrandt, L. and Ellinghaus, D. (2022). EagleImp: fast and accurate genome-wide phasing and imputation in a single tool. *Bioinformatics*, 38(22):4999–5006.

Downloaded from https://academic.oup.com/bioinformaticsadvances/advance-article/doi/10.1093/bioadv/vbad021/7067758 by Fachhochschulen user on 10 March 2023