Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT)

Author: Richard Durbin

Iqbal Group Journal Club

Key Message

Strong compression of haplotype data

Speed up haplotype matching by factor of M

Core Message

Compress haplotype data up to **100 times more** than `gzip`

Core Message

Find all set-maximal matches at position in linear time $O(NM)$ and then for new sequence(s) in $O(N)^*$

* Provided the index array fits in memory

Core Message

Provides efficient framework for seed matching by statistical genotyping algorithms

Section 2.1 - Derivation of prefix arrays

a_k is an array containing the ordering of sequences for the given k

$$\begin{array}{l} 0011010\ 1\ 00 \\ 11\underline{11010}\ 1\ 11 \\ 11111\underline{10}\ 1\ 00 \end{array}$$

So here, $a_7 = [0, 1, 2]$

Whereas $a_9 = [0, 2, 1]$

Section 2.1 - Derivation of prefix arrays

d_k is an array containing the "longest common prefix" at k.

$\texttt{0011010 1 00}$

$\texttt{11}\underline{\texttt{11010}}\texttt{ 1 11}$

$\texttt{11111}\underline{\texttt{10}}\texttt{ 1 00}$

So here, $d_7 = [0, 2, 5]$

Section 2.1 - Derivation of prefix arrays

You can see that at a given k the sequences with the longest match up to $k-1$ will cluster together.

$$\mathtt{0011010\ \ 1\ \ 00}$$
$$\mathtt{11\underline{11010}\ \ 1\ \ 11}$$
$$\mathtt{11111\underline{10}\ \ 1\ \ 00}$$

Section 2.1 - Derivation of prefix arrays

In the below example, y_0 and y_1 are **not** set-maximal.

$$\begin{array}{l}
\texttt{0011010}\ \ \mathbf{1}\ \ \texttt{00} \\
\texttt{11}\underline{\texttt{11010}}\ \ \mathbf{1}\ \ \texttt{11} \\
\texttt{11111}\underline{\texttt{10}}\ \ \mathbf{1}\ \ \texttt{00}
\end{array}$$

The match can be extended…

Section 2.1 - Derivation of prefix arrays

In the below example, y_0 and y_1 are **not** set-maximal.

$$\texttt{00110101}\ \boldsymbol{\texttt{0}}\ \texttt{0}$$
$$\texttt{11}\underline{\texttt{110101}}\ \boldsymbol{\texttt{1}}\ \texttt{1}$$
$$\texttt{11111}\underline{\texttt{101}}\ \boldsymbol{\texttt{0}}\ \texttt{0}$$

The match can be extended…

Section 2.1 - Derivation of prefix arrays

A set-maximal match is like that seen between y_0 and y_1 from $[2, k)$ and there is no y_i with a match longer that includes the interval $[2, k)$

$$\texttt{00110101}\ \boldsymbol{\texttt{0}}\ \texttt{0}$$

$$\texttt{11}\underline{\texttt{110101}}\ \boldsymbol{\texttt{1}}\ \texttt{1}$$

$$\texttt{11111}\underline{\texttt{101}}\ \boldsymbol{\texttt{0}}\ \texttt{0}$$

Section 2.1 - Derivation of prefix arrays

Algorithm 2 - Build prefix and divergent arrays. Update a and d to a_{k+1} and d_{k+1}

```python
u,v = 0
p,q = k+1
a,b,d,e = []
for i in range(M):
        if d_k[i] > p:
                p = d_k[i]
        if d_k[i] > q:
                q = d_k[i]
        if y[i][k] == 0:
                a[u] = a_k[i]
                d[u] = p
                u += 1
                p = 0
        elif y[i][k] == 1:
                b[v] = a_k[i]
                e[v] = q
                v += 1
                q = 0
return a + b, d + e
```

Section 2.2 - Finding all matches within X longer than a minimum length L

Sequences with matches larger than a given L will occur in blocks - separated by blocks where it isn't ($d_k[i] > k - L$)

Therefore just need to keep track of last time $d_k[i] > k - L$

Section 2.2 - Finding all matches within X longer than a minimum length L

This is implemented in **Algorithm 3** and runs in $O[max(NM, matches)]$

Can be done in the same sweep to calculate a and d

Section 2.3 - Finding all set-maximal matches within X in linear time

Given previous methods, for any sequence y_i at position k, it's set-maximal match **must** be immediately prior or following.

Section 2.3 - Finding all set-maximal matches within X in linear time

Find your interval of longest matches at k - Algorithm 3

If $y_j[k] \neq y_i[k]$ then this is set-maximal.

Otherwise, the match can be extended and is therefore not set-maximal.

Runs in $O(NM)$ - Algorithm 4

Section 2.4 - Finding all set-maximal matches from a new sequence z to X

Implemented in **Algorithm 5** is a way to determine the set-maximal matches of a new sequence(s).

This can be done in $O(N)$

I can't quite follow how the algorithm works.

Section 2.5 - Compact representation of X

Storing a_k and d_k for all k would use more space than the original binary data.

Not needed all the time, but are needed if wanting to analyse a new sequence.

Section 2.5 - Compact representation of X

Because there is long stretches of identical values in the sequences the PBWT should be storable in smaller-than-original space.

Could be even more space-efficient by storing only a subset of the a and d arrays, say every 32 or 64 positions.

Makes mention of scope for improving storage of d using Huffman coding.

Results

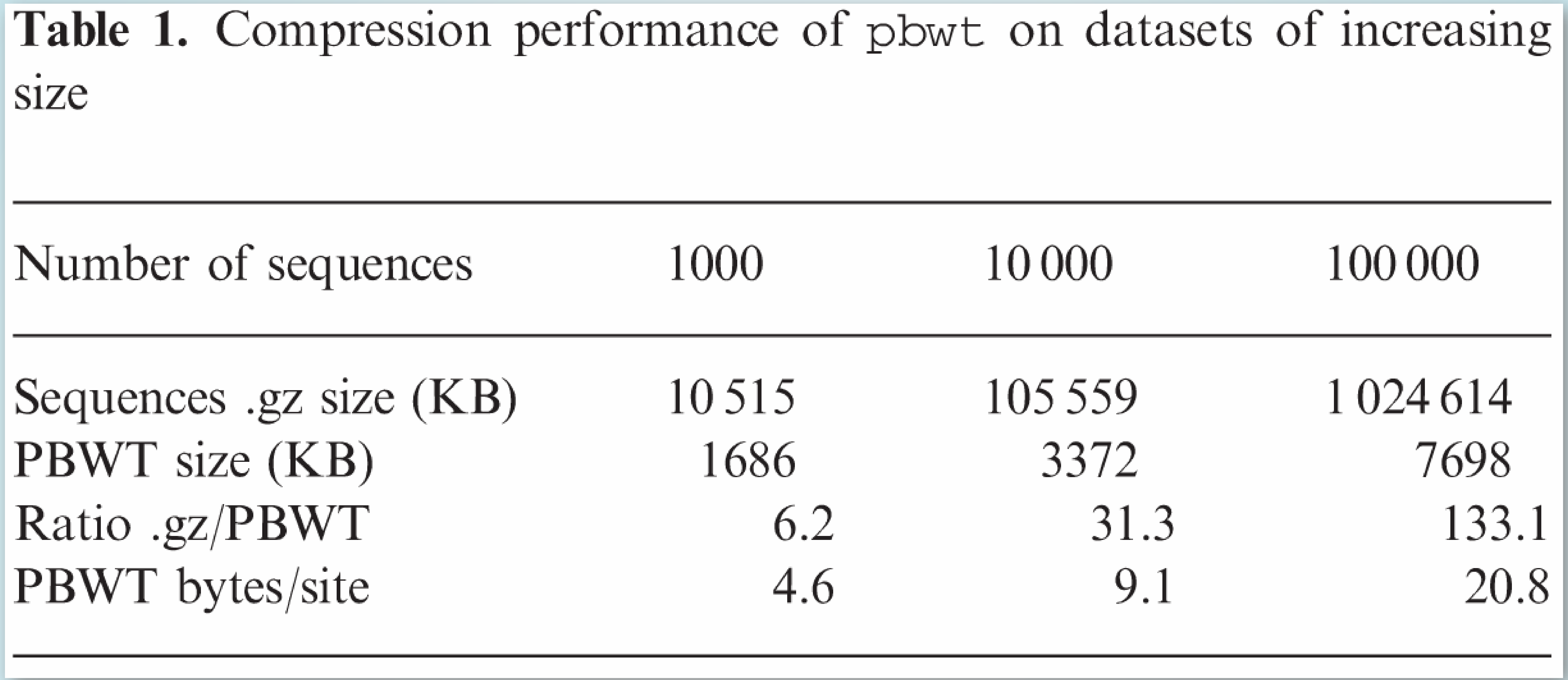

Table 1. Compression performance of `pbwt` on datasets of increasing size

Number of sequences	1000	10 000	100 000
Sequences .gz size (KB)	10 515	105 559	1 024 614
PBWT size (KB)	1686	3372	7698
Ratio .gz/PBWT	6.2	31.3	133.1
PBWT bytes/site	4.6	9.1	20.8

Non-linear benefit for compression as M increases.

Results

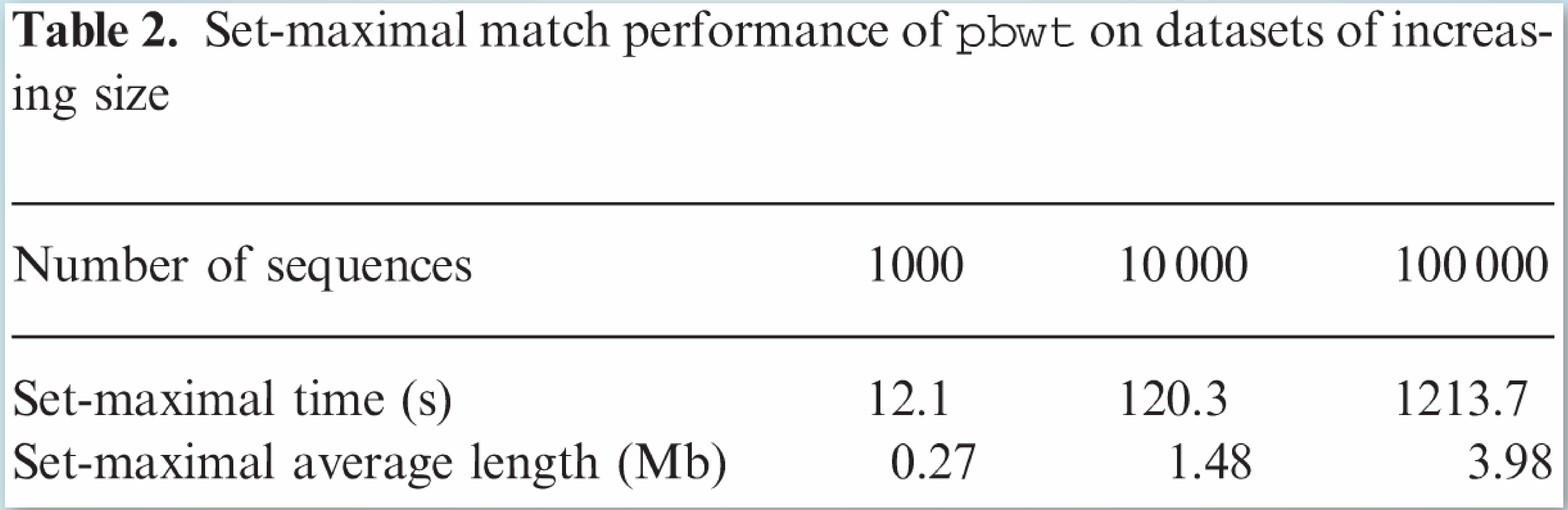

Table 2. Set-maximal match performance of `pbwt` on datasets of increasing size

Number of sequences	1000	10 000	100 000
Set-maximal time (s)	12.1	120.3	1213.7
Set-maximal average length (Mb)	0.27	1.48	3.98

Clearly demonstrates the linear time to find set-maximal matches.

Results

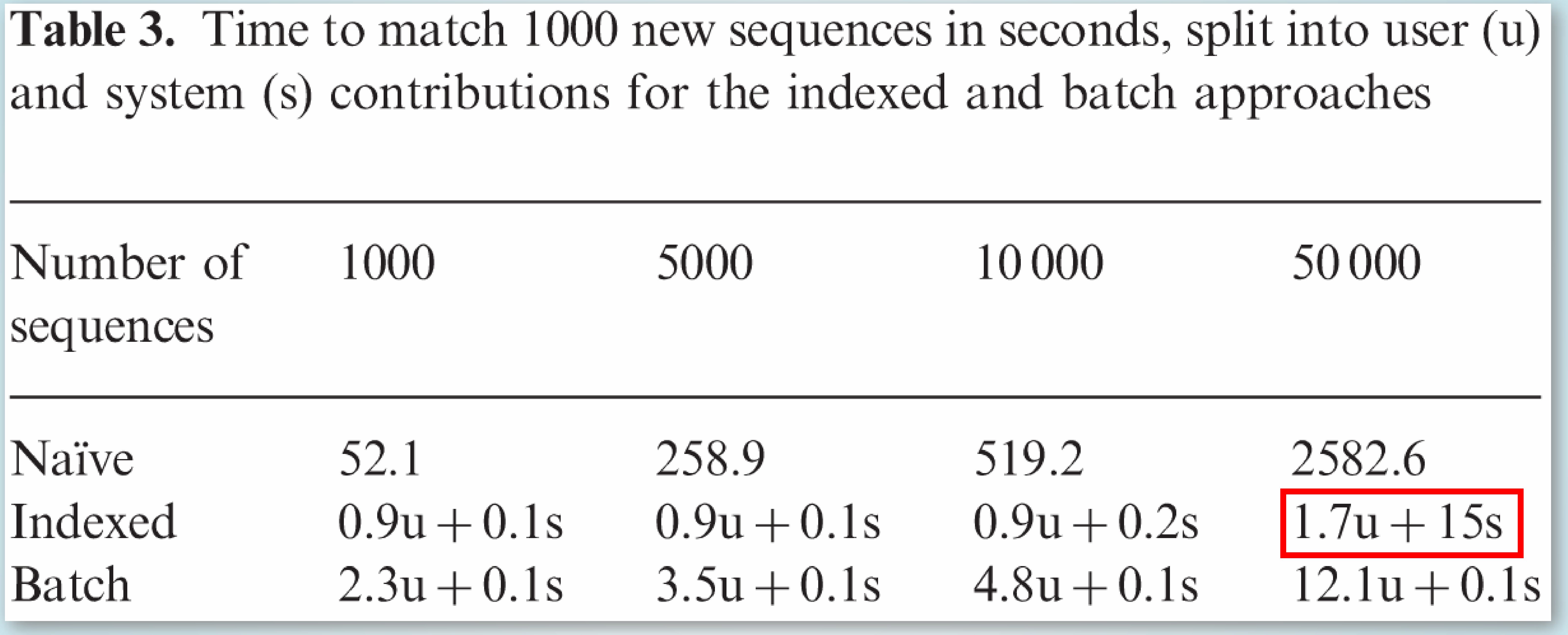

Table 3. Time to match 1000 new sequences in seconds, split into user (u) and system (s) contributions for the indexed and batch approaches

Number of sequences	1000	5000	10 000	50 000
Naïve	52.1	258.9	519.2	2582.6
Indexed	0.9u + 0.1s	0.9u + 0.1s	0.9u + 0.2s	$\boxed{1.7u + 15s}$
Batch	2.3u + 0.1s	3.5u + 0.1s	4.8u + 0.1s	12.1u + 0.1s

Indexed takes constant time up until the point where it runs out of memory.

Results

“I therefore conclude that the PBWT-based approach can be hunreds of times faster than a direct search approach and find matches in time independent of the reference panel size, as conjectured above, so long as the associated index arrays fit in memory.”

Points to Ponder

“Although the algorithms are presented for binary data, they can be extended to multi-allelic data with a little care.”

Points to Ponder

All algorithms require exact matching.

Could use as a seeding method for statistical genotype algorithms.

Mentions possible speedup to BEAGLE

Points to Ponder

More direct approach would be to work from both ends of the sequences.

At some point k we have information about matches in both directions.

Can then assign alleles for all sequences at k before moving to next k

Conclusion

Series of algorithms to generate prefix array data structures for haplotype sequences and for strong compression.

Removes factor of M for search time of new sequences. Allowing matches for ten's of thousands of sequences in minutes.

Conclusion

Importantly, this could provide a way of storing linkage information for a genome graph.

As this can be challenging due to memory limitations.