Burrows-Wheeler Transform for Terabases

Jouni Sirén
Wellcome Trust Sanger Institute

Burrows-Wheeler transform

- Add a unique terminator ($) to the end of the text, sort the suffixes in lexicographic order, and output the preceding character for each suffix.
- Use distinct terminators for multiple texts.
- The permutation is easily reversible and makes the text easier to compress (Burrows & Wheeler, 1994).
- The combinatorial structure is similar to the suffix array, which makes the BWT useful as a space-efficient text index (Ferragina & Manzini, 2000, 2005).

```
 TAGCATAGAC$

C $
G AC$
T AGAC$
T AGCATAGAC$
C ATAGAC$
A C$
G CATAGAC$
A GAC$
A GCATAGAC$
A TAGAC$
$ TAGCATAGAC$
```

Large-scale BWT construction

Strategy: A single BWT is faster to query, while it is easier to build multiple smaller BWTs.

Speed: Should the construction finish overnight? Over the weekend? In two weeks?

Memory: We may have less than n bits of memory for sorting n suffixes.

Hardware: We may not have large amounts of fast disk space, GPUs, or other non-standard hardware.

Efficiency: How many nodes we can afford to use?

Example: Read Server

1000 Genomes Project Phase 3

Low-coverage and exome data: 2535 samples, 922 billion reads, 86 Tbp.

Error correction: A trade-off between losing data and not correcting sequencing errors.

Corrected data: 819 billion reads trimmed to 73 bp or 100 bp, 53.0 billion unique sequences, 4.88 Tbp.

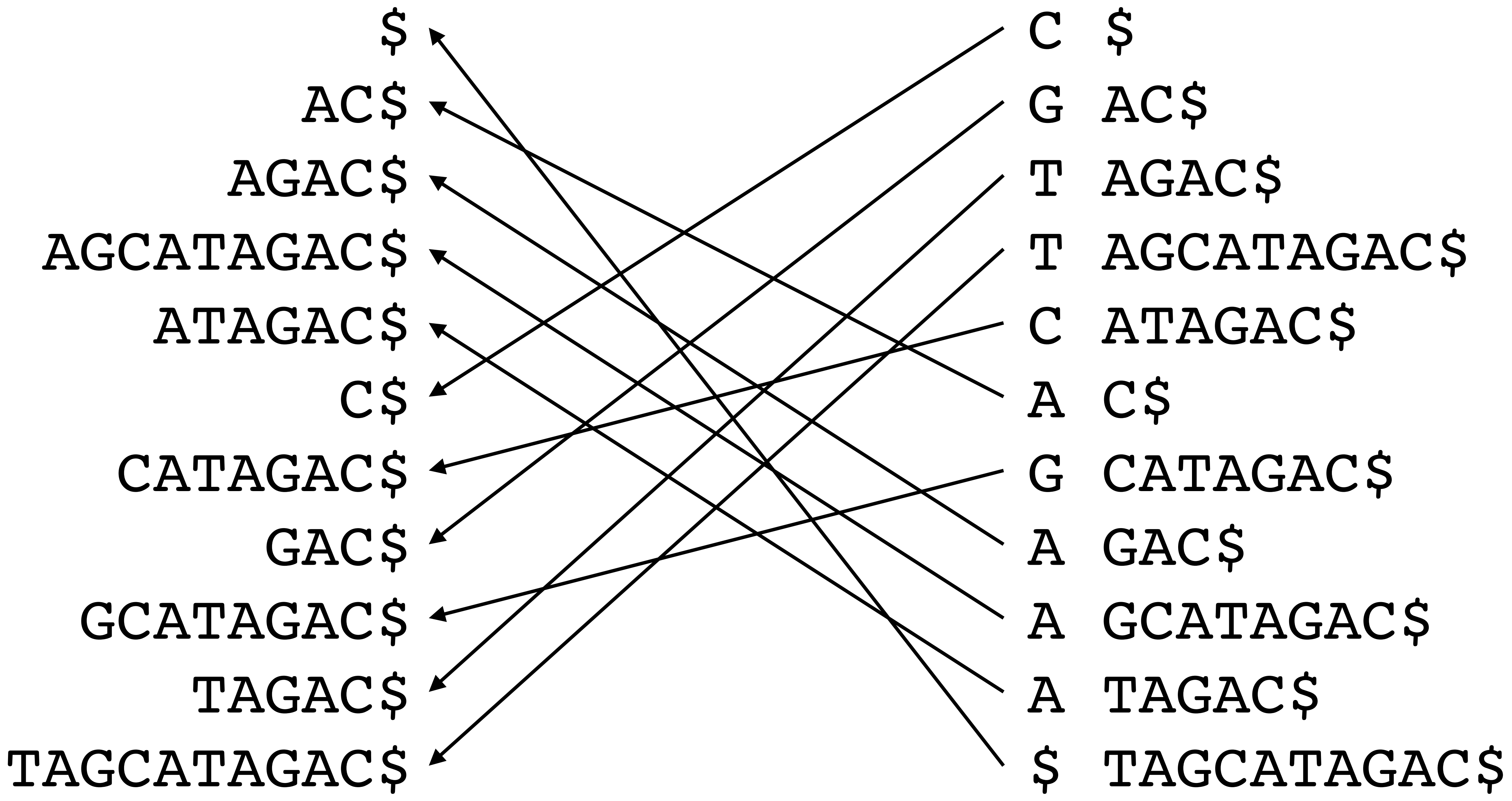
Indexes: 16 run-length encoded BWTs taking 561.5 GB, original read information in metadata databases.

A single node

- 32 CPU cores
- 256 gigabytes of memory
- 369 gigabytes of local disk space at /tmp
- Plenty of shared disk space with no performance guarantees

https://github.com/wtsi-svi/ReadServer

LF-mapping



LF(i) = C[BWT[i]] + BWT.rank(i, BWT[i])

Hypothetical suffixes

```
            $                  C $
          AC$                  G AC$
        AGAC$                  T AGAC$
  AGCATAGAC$      LF(4, c)     T AGCATAGAC$
      ATAGAC$                  C ATAGAC$
           C$                  A C$
     CATAGAC$                  G CATAGAC$
         GAC$                  A GAC$
    GCATAGAC$                  A GCATAGAC$
       TAGAC$                  A TAGAC$
 TAGCATAGAC$                   $ TAGCATAGAC$
```

LF(i, c) = C[c] + BWT.rank(i, c)

TAGCATAGAC$		**C**TAGCATAGAC$	
$	C	$	C
AC$	G	AC$	G
AGAC$	T	AGAC$	T
AGCATAGAC$	T	AGCATAGAC$	T
ATAGAC$	C	ATAGAC$	C
C$	A	C$	A
CATAGAC$	G	CATAGAC$	G
		CTAGCATAGAC$	**$**
GAC$	A	GAC$	A
GCATAGAC$	A	GCATAGAC$	A
TAGAC$	A	TAGAC$	A
TAGCATAGAC$	$	TAGCATAGAC$	**C**

Insert **c** to the beginning:

2. Insert **$** after position LF(i, c).

1. Replace the **$** at position i with the inserted **c**.

Construction options

From a suffix array	In-memory construction uses too much memory. Disk-based algorithms are too slow at 1–2 MB/s.
Direct BWT construction	General-purpose algorithms too slow at 1–2 MB/s; memory issues with large datasets.
DNA-specific algorithms	Reach 5–10 Mbp/s; memory issues beyond 1 Tbp.
GPU-based algorithms	Exotic hardware required; major memory issues.
Distributed algorithms	Efficiency issues; require large amounts of hardware.

Direct BWT construction

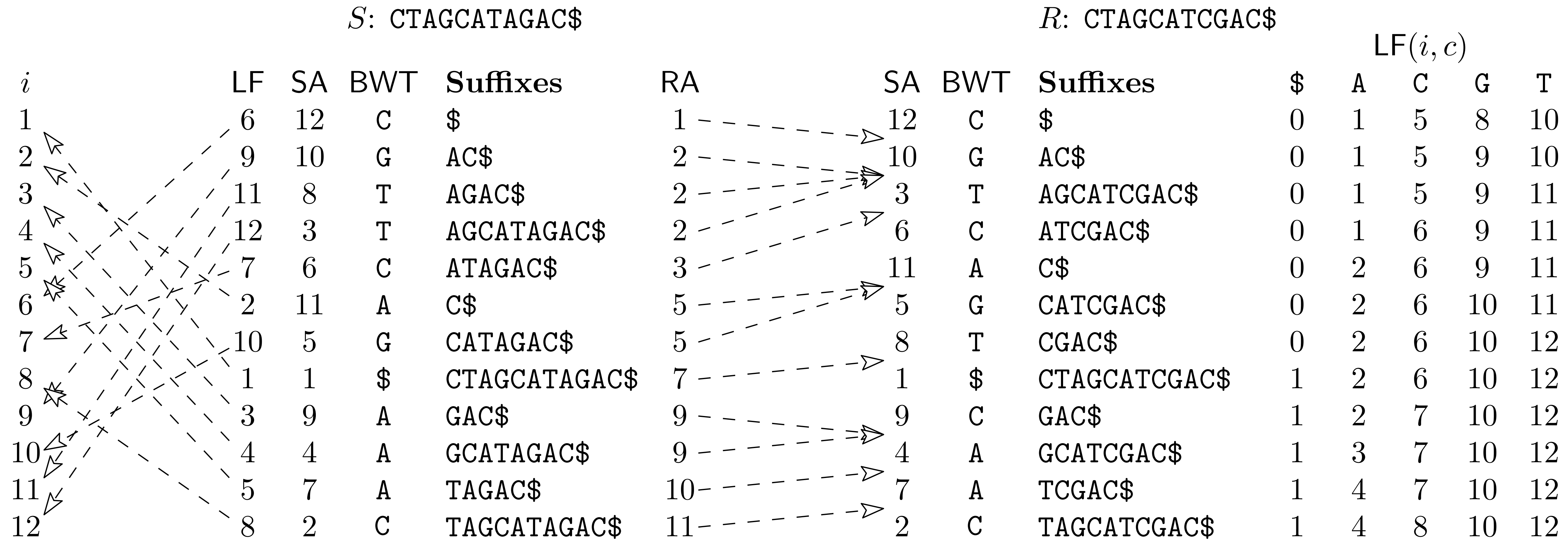
Batch updates: Transform the BWT of S into the BWT of XS. Overhead: O(|X| log |XS|) bits. (Hon et al., 2007)

Dynamic BWT: Store the BWT in a search tree to support edit operations. Size increases by a constant factor, e.g. by 1.5x. (Chan et al., 2007)

Merging algorithm: Merge the BWTs of A and B into the BWT of A∪B. Overhead: min(|A∪B|, |B| log |A|) bits. (Sirén, 2009)

Short reads: Extend all texts at once. Fast version keeps the texts in memory. (Bauer et al., 2013)

BWT merging



S: CTAGCATAGAC$

i	LF	SA	BWT	**Suffixes**	RA
1	6	12	C	$	1
2	9	10	G	AC$	2
3	11	8	T	AGAC$	2
4	12	3	T	AGCATAGAC$	2
5	7	6	C	ATAGAC$	3
6	2	11	A	C$	5
7	10	5	G	CATAGAC$	5
8	1	1	$	CTAGCATAGAC$	7
9	3	9	A	GAC$	9
10	4	4	A	GCATAGAC$	9
11	5	7	A	TAGAC$	10
12	8	2	C	TAGCATAGAC$	11

R: CTAGCATCGAC$

			LF(i,c)				
SA	BWT	**Suffixes**	$	A	C	G	T
12	C	$	0	1	5	8	10
10	G	AC$	0	1	5	9	10
3	T	AGCATCGAC$	0	1	5	9	11
6	C	ATCGAC$	0	1	6	9	11
11	A	C$	0	2	6	9	11
5	G	CATCGAC$	0	2	6	10	11
8	T	CGAC$	0	2	6	10	12
1	$	CTAGCATCGAC$	1	2	6	10	12
9	C	GAC$	1	2	7	10	12
4	A	GCATCGAC$	1	3	7	10	12
7	A	TCGAC$	1	4	7	10	12
2	C	TAGCATCGAC$	1	4	8	10	12

BWT$_{RS}$	C	C	G	G	T	T	T	C	C	A	A	G	G	T	$	$	C	A	A	A	A	A	C	C
Source	R	S	R	S	S	S	R	S	R	R	S	S	R	R	S	R	R	S	S	R	S	R	S	R
B_{RS}	0	1	0	1	1	1	0	1	0	0	1	1	0	0	1	0	0	1	1	0	1	0	1	0

We start with RA[1] = 1 and then iterate RA[LF$_S$(i)] = LF$_R$(RA[i], BWT$_S$[i]).

BWT merging

Search: Generate the ranks in any order. There are many possible low-level optimizations. Multiple threads can be used to handle different sequences.

Sort: Sort the ranks to build the rank array or the interleaving bitvector. This can be done in parallel with the other phases.

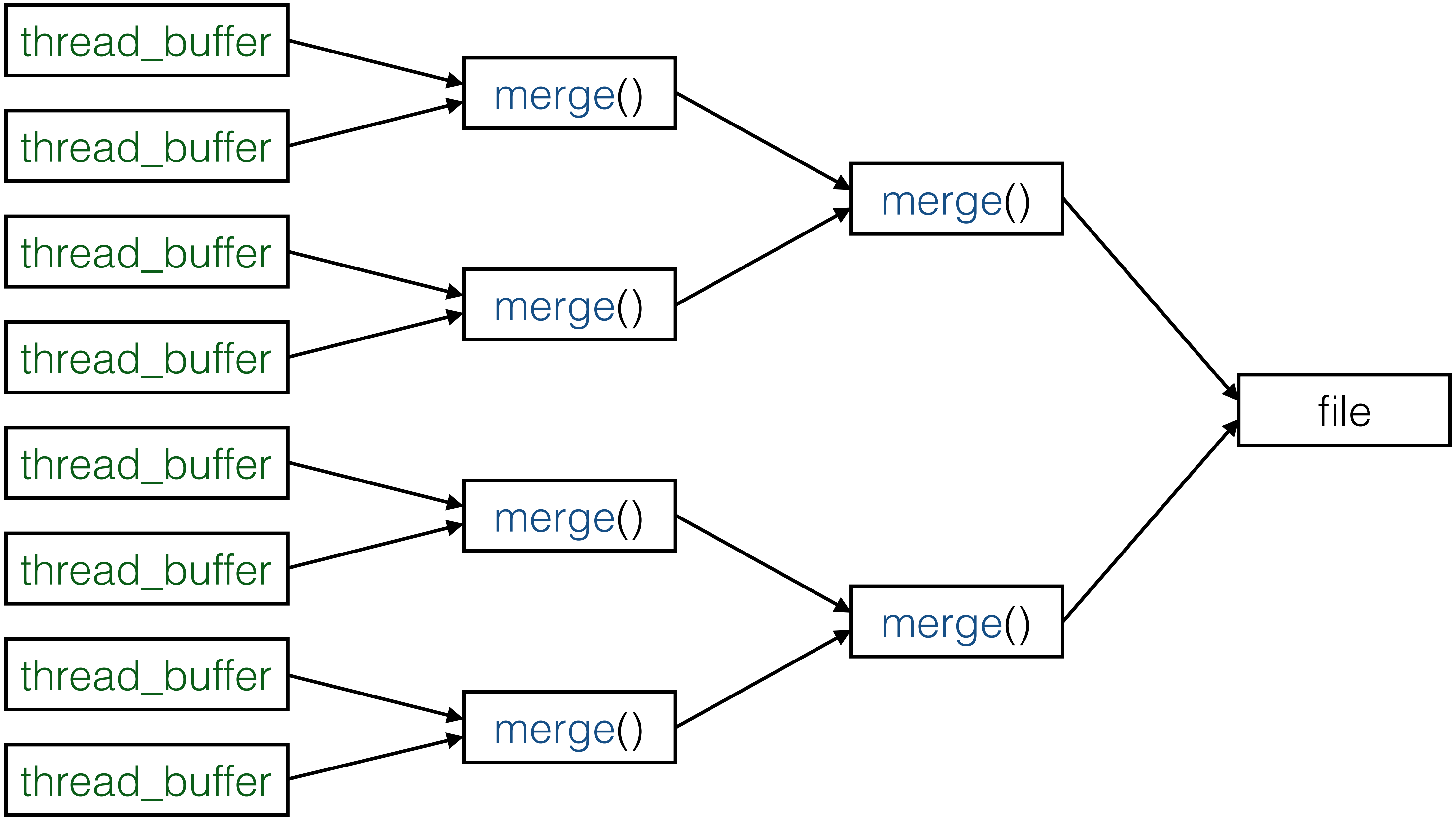
Merge: Interleave the source BWTs according to the rank array or the interleaving bitvector. This can be done almost in-place with block-based arrays.

Search thread

```
while …
    (rank, count) ← nextRun()
    run_buffer.insert(rank, count)
    if run_buffer.full():
        sort(run_buffer)
        compress(run_buffer)
        thread_buffer ← merge(run_buffer, thread_buffer)
        if thread_buffer.full():
            merge(thread_buffer, merge_buffers)
```

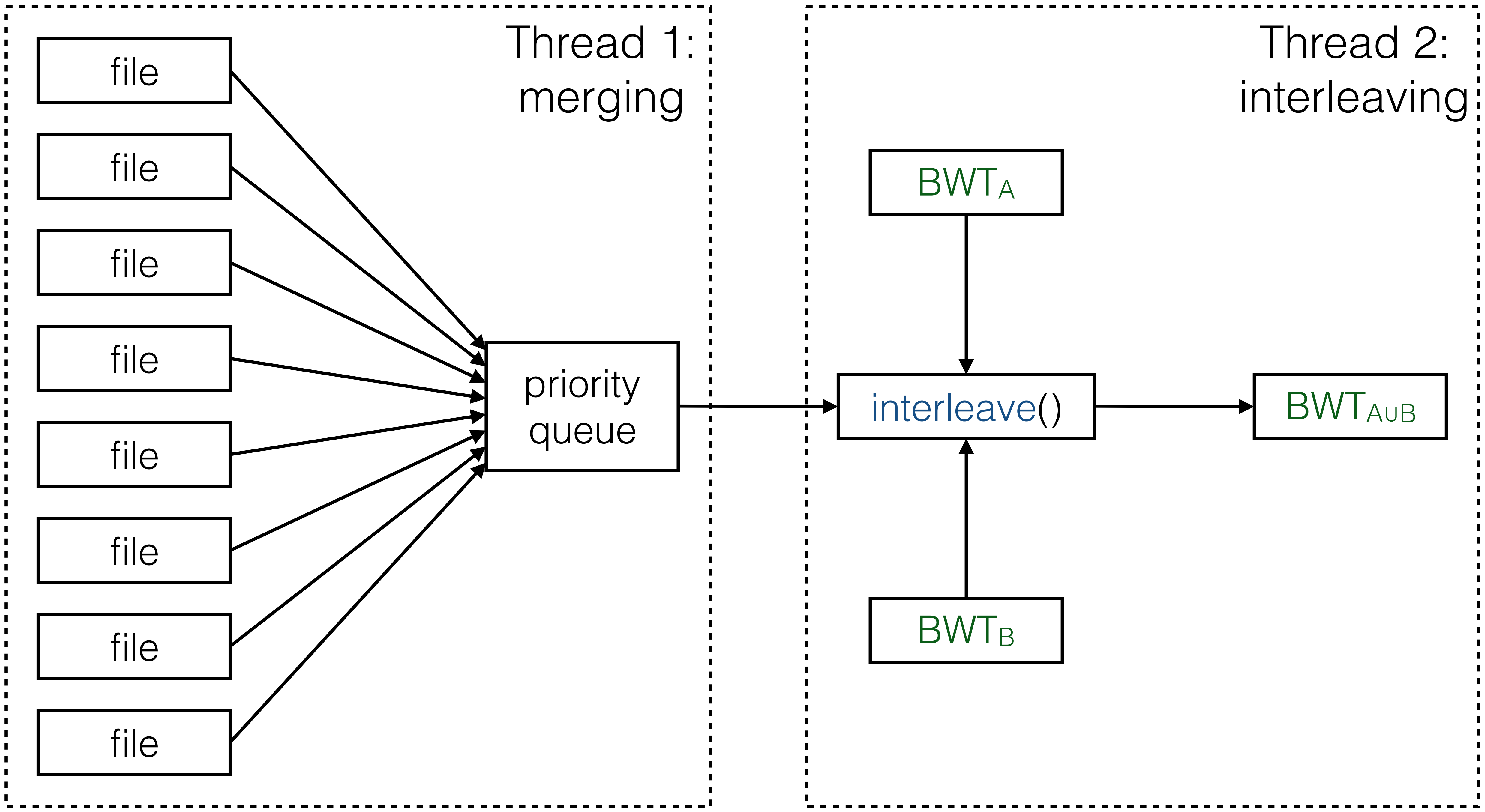
compress() uses **differential encoding** for the ranks and byte-level **prefix-free codes** for encoding the stream of rank differences and run lengths.

Merge buffers



We need one **merge buffer** / level.

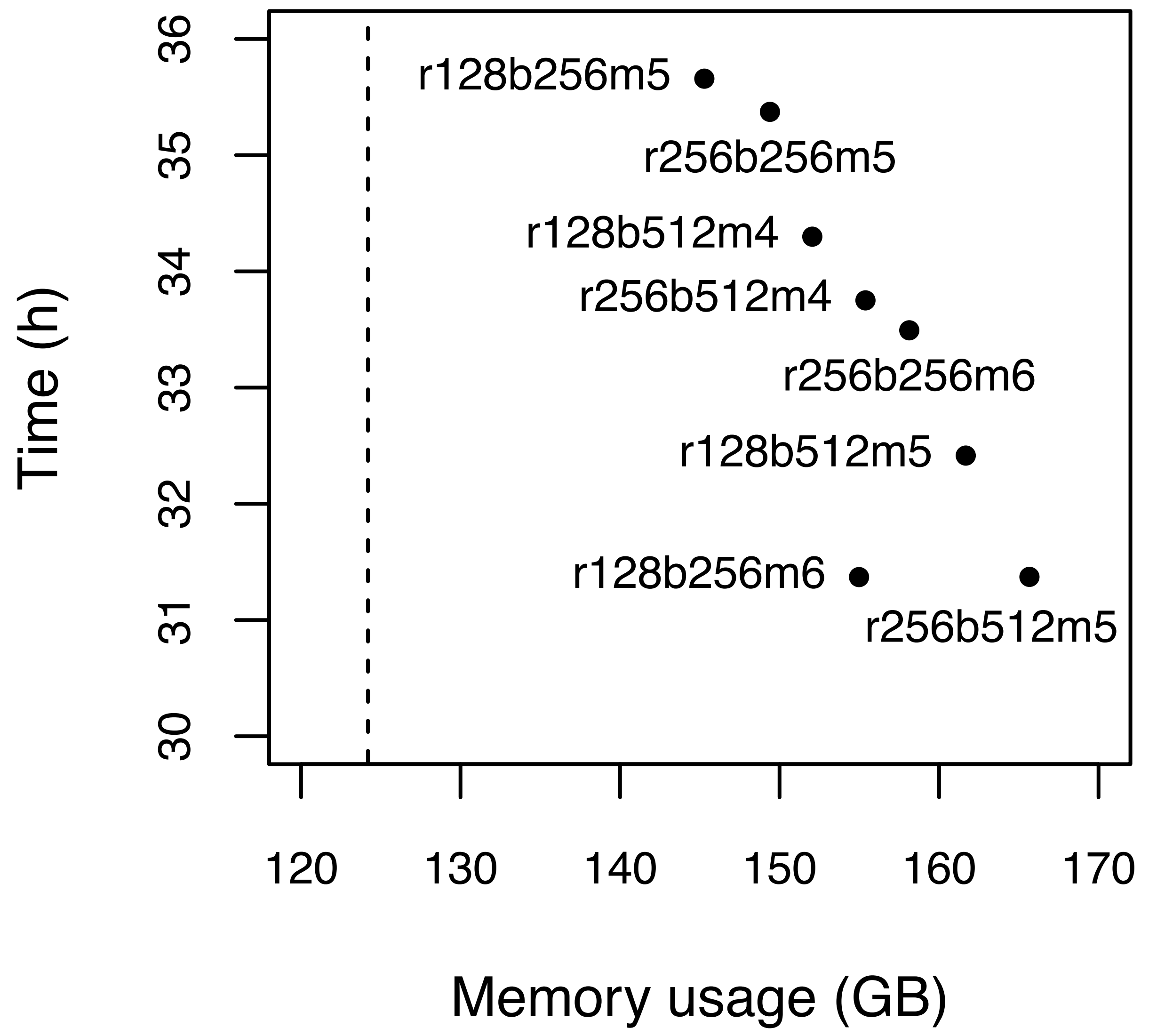
Merge phase



Multithreaded merging would help with a faster disk.

Dataset	Reads	Size
Read Server: AA, TT, AT, TA	16.2 billion	1.49 Tbp
CEU trio: NA12878, NA12891, NA12892	7.63 billion	771 Gbp
Read Server: *A, *C	26.5 billion	2.45 Tbp
Read Server: *G, *T	26.5 billion	2.44 Tbp

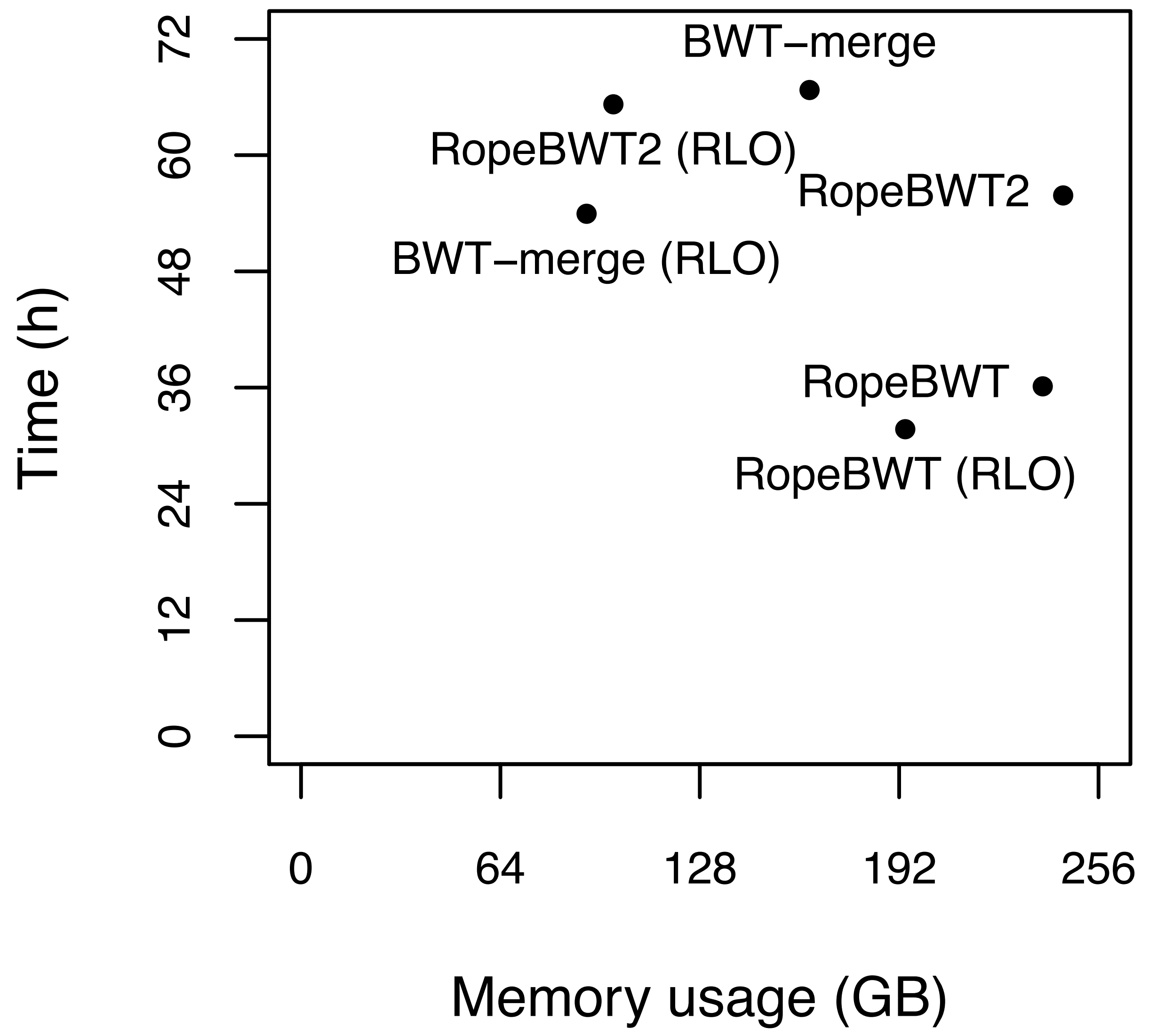
Read Server: AA, TT, AT, TA



rXbYmZ: X MB run buffers, Y MB thread buffers, Z merge buffers.

r128b256m6 had the best performance: 9.40 Mbp/s throughput, 30.8 GB memory overhead.

CEU trio



RopeBWT: The algorithm of Bauer et al. for short reads.

RopeBWT2: Dynamic FM-index. Memory usage: ~1.5·|BWT| + 15 GB.

BWT-merge: Merge the BWTs build by RopeBWT. Uses ~|BWT| + 30 GB of memory.

Read Server: *A, *C / *G, *T

	Individual BWTs	Merged BWTs
Read Server format	281 GB 281 GB	239 GB 239 GB
BWT-merge format	225 GB 226 GB	181 GB 180 GB

81.3 hours
221 GB memory
297 GB disk

83.0 hours
219 GB memory
300 GB disk

Conclusions

- We can merge BWTs at 600 – 800 Gbp/day with 30 gigabytes of memory overhead.
- This makes it possible to build multi-terabase BWTs on the systems they will be used.
- Merging the Read Server BWTs into two files and converting them to the new format reduced their size from 560 GB to 360 GB.
- https://github.com/jltsiren/bwt-merge