Burrows-Wheeler transform for terabases

Jouni Sirén

Wellcome Trust Sanger Institute
Wellcome Genome Campus
Hinxton, Cambridge, CB10 1SA, UK
jouni.siren@iki.fi

Abstract

In order to avoid the reference bias introduced by mapping reads to a reference genome, bioinformaticians are investigating reference-free methods for analyzing sequenced genomes. With large projects sequencing thousands of individuals, this raises the need for tools capable of handling terabases of sequence data. A key method is the Burrows-Wheeler transform (BWT), which is widely used for compressing and indexing reads. We propose a practical algorithm for building the BWT of a large read collection by merging the BWTs of subcollections. With our 2.4 Tbp datasets, the algorithm can merge 600 Gbp/day on a single system, using 30 gigabytes of memory overhead on top of the run-length encoded BWTs.

Introduction

The decrease in the cost of DNA sequencing has flooded the world with *sequence data*. The *1000 Genomes Project* [1] sequenced the genomes of over 2500 humans, and there are other projects that are similar or greater in scale. A sequencing machine produces a large number of *reads* (short sequences) that cover the genome many times over. For a 3 Gbp human genome, the total length of the reads is often 100 Gbp or more.

De novo assembly of sequenced genomes is still too difficult to be routinely done. As a practical alternative, bioinformaticians usually align the reads to a *reference genome* of the same species. Because most reference genomes come from the genomes of a small number of individuals, this introduces *reference bias*, which may adversely affect the results of subsequent analysis. Switching from reference sequences to *reference graphs* can reduce the bias, but such transition will likely take years [2].

Preprocessing large datasets can take weeks. It is often not feasible to rebuild everything when new methods of analysis require new functionalities. Structures based on the *Burrows-Wheeler transform* (BWT) are often useful due to their versatility. A *run-length encoded* BWT compresses repetitive sequence collections quite well [3], while the similarities to the suffix tree and the suffix array make BWT-based indexes suitable for many *pattern matching* and *sequence analysis* tasks [4, 5].

The *Read Server* project at the Sanger Institute develops tools for large-scale *reference-free* genome analysis, avoiding reference bias. Unique reads are compressed and indexed using the BWT, while metadata databases contain information on the original reads. Initially, the project works with data from the 1000 Genomes Project. After error correction and trimming the reads to either 73 bp or 100 bp, the 922 billion original reads (86 Tbp) are reduced to 53.0 billion unique sequences (4.88 Tbp). These sequences are stored in 16 BWT-based indexes [6] taking a total of 561.5 gigabytes.

This work was supported by the Wellcome Trust grant [098051].

The unique reads are partitioned between the BWTs by the last two bases. Every query must be repeated in all 16 indexes. The BWTs also require more space, as we cannot compress the similarities between the reads in different indexes. Reducing the number of indexes would improve both memory usage and query performance. This requires BWT construction algorithms that can handle terabases of data.

There are four often contradictory requirements for large-scale BWT construction: **Speed.** Larger datasets require faster algorithms. As a rough guideline, an algorithm processing 1 Mbp/s is good for up to 100 Gbp, while remaining somewhat useful until 1 Tbp of data. **Memory.** We may have to process n bp datasets on systems with less than n bits of memory. **Hardware.** A single node in a typical computer cluster has tens of CPU cores, from tens to hundreds of gigabytes of memory, a limited amount of local disk space, and access to shared disk space with no performance guarantees. Algorithms using a GPU or a large amount of fast disk space require special-purpose hardware. **Efficiency.** Large BWTs can be built by doing a lot of redundant work on multiple nodes. As most computer clusters do not have large amounts of unused capacity, such inefficient algorithms are not suitable for repeated use.

The most straightforward approach to BWT construction is to build a *suffix array* using a fast general-purpose algorithm [7, 8], and then derive the BWT from the suffix array. These algorithms cannot be used with large datasets, as they require much more memory than the sequences themselves. Suffix arrays can be built on disk [9], but even the fastest algorithms cannot index the data faster than 1–2 Mbp/s [10].

There are many *direct* BWT construction algorithms that do not need the suffix array. Some require a limited amount of working space on top of the BWT [11, 12, 13, 14], while others use the disk as additional working space [15, 16]. These general-purpose algorithms rarely exceed 1–2 Mbp/s. *Specialized algorithms* for DNA sequences achieve better time/space trade-offs. Some can index 5–10 Mbp/s using ordinary hardware, with their memory usage becoming the bottleneck after about 1 Tbp [17, 18]. GPU-based algorithms are even faster, but their memory usage is also higher [19, 20]. Distributing the BWT construction to multiple nodes can remove the obvious bottlenecks, at the price of using more resources for the construction [21].

In this paper, we propose a practical algorithm for building the BWT for terabases of sequence data. The algorithm is based on dividing the sequence collection into a number of subcollections, building the BWT for each subcollection, and *merging* the BWTs into a single structure [13]. The merging algorithm is faster than BWT construction for the subcollections, while having a relatively small memory overhead on top of the final BWT-based index. As the index must be loaded in memory for use, it can be built on the same system as it is going to be used.

Background

A *string* $S[1, n] = s_1 \cdots s_n$ is a sequence of *characters* over an *alphabet* $\Sigma = \{1, \ldots, \sigma\}$. For indexing purposes, we consider *text* strings $T[1, n]$ terminated by an endmarker $T[n] = \$ = 0$ not occurring elsewhere in the text. *Binary* sequences are strings over the alphabet $\{0, 1\}$. A *substring* of string S is a sequence of the form $S[i, j] = s_i \cdots s_j$. We call substrings of the type $S[1, j]$ and $S[i, n]$ *prefixes* and *suffixes*, respectively.

The *suffix array* (SA) [22] is a simple full-text index. Given a text T, its suffix array $\mathsf{SA}_T[1,n]$ is an array of pointers to the suffixes of the text in *lexicographic order*.[1] We can build the suffix array in $\mathsf{O}(n)$ time using $2n$ bits of working space on top of the text and the suffix array [8]. Given a *pattern* P, we can find the *lexicographic range* $[sp, ep]$ of suffixes prefixed by the pattern in $\mathsf{O}(|P| \log n)$ time. The range of pointers $\mathsf{SA}[sp, ep]$ lists the *occurrences* of the pattern in the text.

The suffix array requires several times more memory than the original text. For large texts, this can be a serious drawback. We can use the *Burrows-Wheeler transform* (BWT) [23] as a more space-efficient alternative to the suffix array. The BWT is an easily reversible permutation of the text with a similar combinatorial structure to the suffix array. Given a text $T[1,n]$ and its suffix array, we can easily produce the BWT as $\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$ (with $\mathsf{BWT}[i] = T[n]$, if $\mathsf{SA}[i] = 1$).

If $X \leq Y$ in lexicographic order, we also have $cX \leq cY$ for any character c. If $\mathsf{BWT}[i] = c$ is the j-th occurrence of c in the BWT and $\mathsf{SA}[i]$ points to suffix X, suffix cX is the j-th suffix starting with c in lexicographic order.

Let $\mathsf{C}[c]$ bet the number of suffixes starting with a character smaller than c, and let $S.\mathsf{rank}(i,c)$ be the number of occurrences of c in the prefix $S[1,i]$. We define LF-*mapping* as $\mathsf{LF}(i,c) = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}(i,c)$ and $\mathsf{LF}(i) = \mathsf{LF}(i, \mathsf{BWT}[i])$. The general form $\mathsf{LF}(i,c)$ is the number of suffixes X of text T with $X \leq cT[\mathsf{SA}[i], n]$. This is known as the *lexicographic rank* $\mathsf{rank}(cT[\mathsf{SA}[i], n], T)$ of text $cT[\mathsf{SA}[i], n]$ among the suffixes of text T. The specific form $\mathsf{LF}(i)$ gives the lexicographic rank of the previous suffix ($\mathsf{SA}[\mathsf{LF}(i)] = \mathsf{SA}[i] - 1$, if $\mathsf{SA}[i] > 1$, and $\mathsf{SA}[\mathsf{LF}(i)] = n$ otherwise).

The *FM-index* (FMI) [6] is a full-text index based on the BWT. We use *backward searching* in the FM-index to find the lexicographic range $[sp, ep]$ matching pattern P. Let $[sp_i, ep_i]$ be the range of suffixes of text T matching suffix $P[i, |P|]$ of the pattern. We find $[sp_{i-1}, ep_{i-1}]$ as $[\mathsf{LF}(sp_i - 1, P[i-1]) + 1, \mathsf{LF}(ep_i, P[i-1]))]$. By starting from $[sp_{|P|}, ep_{|P|}] = [\mathsf{C}[P[|P|]] + 1, \mathsf{C}[P[|P|] + 1]]$, we can find the lexicographic range of suffixes starting with the pattern in $\mathsf{O}(|P| \cdot t_r)$ time, where t_r is the time required to answer rank queries on the BWT. In practice, the time complexity ranges from $\mathsf{O}(|P|)$ to $\mathsf{O}(|P| \log n)$, depending on the encoding of the BWT.

The FM-index *samples* some suffix array pointers, including the one to the beginning of the text. When unsampled pointers are needed, they are derived by using LF-mapping. If $\mathsf{SA}[i]$ is not sampled, the FM-index proceeds to $\mathsf{LF}(i)$ and continues from there. If $\mathsf{SA}[\mathsf{LF}^k(i)]$ is the first sample encountered, $\mathsf{SA}[i] = \mathsf{SA}[\mathsf{LF}^k(i)] + k$.

Assume that we have an ordered *collection* of texts $\mathcal{A} = (T_1, \ldots, T_m)$ of total length $n = |\mathcal{A}| = \sum_i |T_i|$. We want to build a (generalized) BWT for the collection. The usual way is to make all endmarkers distinct, giving the one at the end of text T_i character value $(0, i)$. This also makes all suffixes of the collection distinct. To save space, we still encode each endmarker as a 0 in the BWT. Because of this, LF-mapping does not work with $c = 0$, and we cannot match patterns spanning text boundaries.

When the texts are short (e.g. reads), there are more space-efficient alternatives to sampling. Because all endmarkers have distinct values during sorting, we know that $\mathsf{SA}[i]$ with $i \leq m$ points to the end of text T_i. To find the end, we iterate $\Psi(i) =$

[1] If the text is evident from the context, we will omit the subscript and write just SA, BWT, etc.

$\mathsf{BWT}.\mathsf{select}(i - \mathsf{C}[c], c)$, where c is the largest value with $\mathsf{C}[c] < i$ and $S.\mathsf{select}(i, c)$ finds the i-th occurrence of character c in string S. If $k \geq 0$ is the smallest value for which $j = \Psi^k(i) \leq m$, we know that $\mathsf{SA}[i]$ points to offset $|T_j| - k$ in text T_j.

We can *extract* text T_i in $\mathsf{O}(|T_i| \cdot t_r)$ time by using LF-mapping [23]. We start from the endmarker at $\mathsf{BWT}[i]$ and extract the text backwards as $T_i[|T_i| - j] = \mathsf{BWT}[\mathsf{LF}^{j-1}(i)]$, for $1 \leq j \leq |T_i|$. As $\mathsf{SA}[\mathsf{LF}^j(i)]$ points to suffix $T_i[|T_i| - j, |T_i|]$, we also find the lexicographic ranks of all suffixes of text T_i in the process.

Space-efficient BWT construction

The FM-index was introduced as a more space-efficient alternative to the suffix array. If we need the suffix array to build the FM-index, a large part of this benefit is lost, and index construction becomes the bottleneck. To overcome the bottleneck, we can use *incremental construction algorithms* that build the FM-index directly. Some of them use an adjustable amount of working space on top of the FM-index, making it possible to index text collections larger than the size of the memory.

Assume that we have built the BWT of text T, and we want to *transform* the BWT into that of text cT, where c is a character [11]. We find the pointer $\mathsf{SA}[i]$ to the beginning of text T (where $\mathsf{BWT}[i] = 0$). Then we determine the lexicographic rank $j = \mathsf{rank}(cT, T) = \mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}(i, c)$ of text cT among the suffixes of text T. Finally we *replace* the endmarker at $\mathsf{BWT}[i]$ with the inserted character c and *insert* a new endmarker between $\mathsf{BWT}[j]$ and $\mathsf{BWT}[j + 1]$.

We can use the transformation for BWT construction in several ways. We can use *batch updates* and transform the BWT of text T into that of text XT, where X is a string [11]. We can start with the BWTs of text collections $\mathcal{A}$ and $\mathcal{B}$, and *merge* them into the BWT of collection $\mathcal{A} \cup \mathcal{B}$ [13]. We can also *extend* multiple texts at once by inserting a new character to the beginning of each of them [17]. In all cases, we can use either *static* or *dynamic* [24] structures for the BWT. Dynamic representations increase the size of the BWT (e.g. by around 1.5x in $\mathsf{RopeBWT2}$ [18]), while static representations require more space overhead for buffering the updates.

Assume that we want to merge the BWTs of two text collections $\mathcal{A}$ and $\mathcal{B}$ of total length $n_{\mathcal{A}}$ and $n_{\mathcal{B}}$, respectively [13]. We store the BWTs in two-level arrays, where the first level contains pointers to b-bit *blocks*. If a BWT takes x bits, the space overhead from the array is $\frac{x}{b} \log x + \mathsf{O}(b)$ bits. This becomes $\mathsf{O}(\sqrt{x \log x})$ bits with $b = \sqrt{x \log x}$. The merging algorithm has three phases: search, sort, and merge. It uses $\mathsf{O}(n_{\mathcal{A}} + n_{\mathcal{B}} t_r)$ time and $\min(n_{\mathcal{B}} \log n_{\mathcal{A}}, n_{\mathcal{A}} + n_{\mathcal{B}}) + \mathsf{O}(\sqrt{x \log x})$ bits of working space in addition to the BWTs and the structures required to use them as FM-indexes. See Figure 1 for an example with two texts.

Search. We search for all texts of collection $\mathcal{B}$ in $\mathsf{BWT}_{\mathcal{A}}$, and output the lexicographic rank $\mathsf{rank}(X, \mathcal{A})$ for each suffix X of $\mathcal{B}$. This takes $\mathsf{O}(n_{\mathcal{B}} t_r)$ time. We either need the collection in plain form, or extract the texts from $\mathsf{BWT}_{\mathcal{B}}$ in the same asymptotic time.

Sort. We build the *rank array* (RA) of $\mathcal{B}$ relative to $\mathcal{A}$ by sorting the ranks. The rank array is defined as $\mathsf{RA}_{\mathcal{B}|\mathcal{A}}[i] = \mathsf{rank}(X, \mathcal{A})$, where $\mathsf{SA}_{\mathcal{B}}[i]$ points to suffix X. The array requires $n_{\mathcal{B}} \log n_{\mathcal{A}}$ bits of space, and we can build it in $\mathsf{O}(sort(n_{\mathcal{B}}, n_{\mathcal{A}}))$ time, where $sort(n, u)$ is the time required to sort n integers from universe $[0, u]$. If we extracted

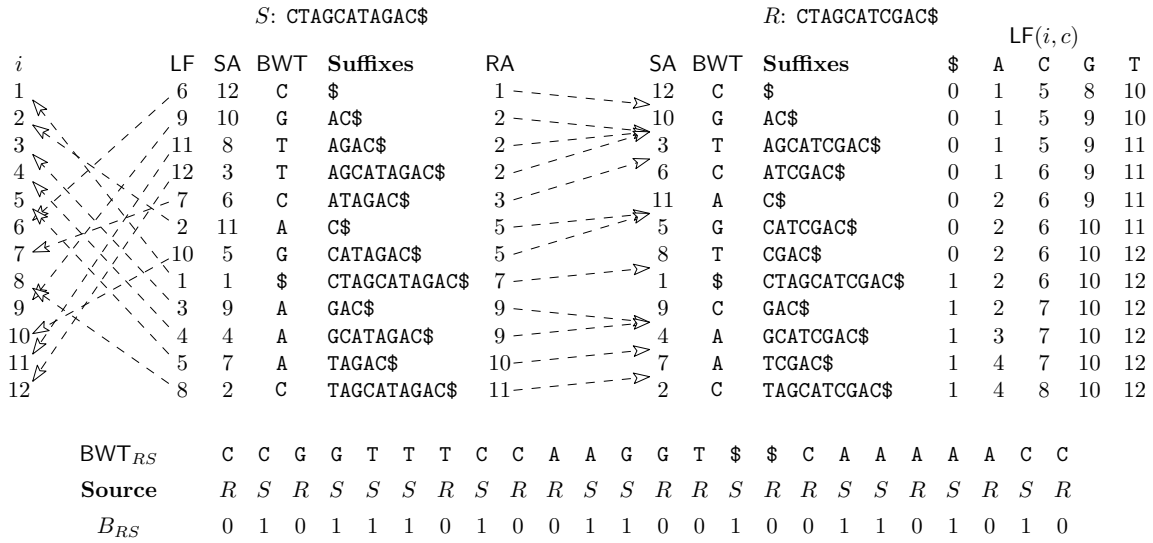

Figure 1: Merging the BWTs of texts R and S. Rank array RA counts the number of suffixes of R that are lexicographically smaller than or equal to the given suffix of S. We fill it by starting with $\mathsf{RA}[1] = 1$ and iterating $\mathsf{RA}[\mathsf{LF}_S(i)] = \mathsf{LF}_R(\mathsf{RA}[i], \mathsf{BWT}_S[i])$. Interleaving bitvector B_{RS} tells whether the source of a character in the merged BWT is in BWT_R or BWT_S. We build it by setting bits $i + \mathsf{RA}[i]$ to 1 for all i.

the texts from $\mathsf{BWT}_\mathcal{B}$, we can write the ranks directly into the rank array, making this phase trivial. We can also encode the rank array as a binary sequence $B_{\mathcal{A}\cup\mathcal{B}}$ of length $n_\mathcal{A} + n_\mathcal{B}$. This *interleaving bitvector* is built by setting $B_{\mathcal{A}\cup\mathcal{B}}[i + \mathsf{RA}_{\mathcal{B}|\mathcal{A}}[i]] = 1$ for $1 \leq i \leq n_\mathcal{B}$. If $B_{\mathcal{A}\cup\mathcal{B}}[j] = 1$, we know that $\mathsf{SA}_{\mathcal{A}\cup\mathcal{B}}[j]$ points to a suffix of $\mathcal{B}$.

Merge. We interleave $\mathsf{BWT}_\mathcal{A}$ and $\mathsf{BWT}_\mathcal{B}$ according to the rank array. If $\mathsf{RA}_{\mathcal{B}|\mathcal{A}}[i] = j$, the merged BWT will have j characters from $\mathsf{BWT}_\mathcal{A}$ before $\mathsf{BWT}_\mathcal{B}[i]$. This phase takes $\mathsf{O}(n_\mathcal{A} + n_\mathcal{B})$ time. By reusing the blocks of $\mathsf{BWT}_\mathcal{A}$ and $\mathsf{BWT}_\mathcal{B}$ for $\mathsf{BWT}_{\mathcal{A}\cup\mathcal{B}}$, we can merge the BWTs almost in-place. The total working space is $\mathsf{O}(\sqrt{x \log x})$ bits, where x is the maximum of the sizes of $\mathsf{BWT}_\mathcal{A}$ and $\mathsf{BWT}_\mathcal{B}$ in bits.

Large-scale BWT merging

If we split a text collection $\mathcal{A}$ of total length n into p *subcollections* of equal size, we can build $\mathsf{BWT}_\mathcal{A}$ incrementally by merging the BWTs of the subcollections. This takes $\mathsf{O}((p + t_r)n)$ time and uses essentially $\min(\frac{n}{p} \log n, n)$ bits of working space.

When the collection is large, the space overhead of the construction algorithm often determines whether we can build the BWT. We can reduce the overhead by writing the lexicographic ranks to *disk*. If we sort the ranks on disk, we just need to scan the rank array once during the merge phase. We can also *compress* the ranks before writing them to disk and *interleave* the sorting with the search and merge phases. We now describe the key ideas for fast and space-efficient BWT construction.

Search. Instead of searching for every text in collection $\mathcal{B}$ separately, we can search for the *reverse trie* of the collection. Assume that there are $m_\mathcal{A}$ texts in collection $\mathcal{A}$ and $m_\mathcal{B}$ texts in collection $\mathcal{B}$. The *root* of the trie corresponds to suffix \$, which has lexicographic rank $m_\mathcal{A}$ in $\mathcal{A}$ and corresponds to lexicographic range $[1, m_\mathcal{B}]$ in $\mathcal{B}$.

Assume that we have a *node* of the trie corresponding to suffix X, lexicographic rank r, and lexicographic range $[sp, ep]$. As suffix X occurs $ep + 1 - sp$ times in collection $\mathcal{B}$, we can output a *run* of ranks $(r, ep+1-sp)$. Afterwards, we proceed to the *children* of the node. For each character $c \in \Sigma$, we create a node corresponding to suffix cX, rank $\mathsf{LF}_{\mathcal{A}}(r, c)$, and range $[\mathsf{LF}_{\mathcal{B}}(sp-1, c)+1, \mathsf{LF}_{\mathcal{B}}(ep, c)]$. Searching the branches of the trie can be done in parallel using multiple *threads*.

Buffering. To reduce disk I/O and space usage, we buffer and compress the lexicographic ranks before writing them to disk. Each thread has two buffers: a *run buffer* and a *thread buffer*. The run buffer stores the runs as pairs of integers (r, ℓ). Once the run buffer becomes full, we sort the runs by *run heads* r, use *differential encoding* for the run heads, and encode the differences and run lengths with a *prefix-free code*. The compressed run buffer is then merged with the similarly compressed thread buffer.

Once the thread buffer becomes full, we merge it with the global *merge buffers*. There are k merge buffers M_1 to M_k, with buffer M_i containing 2^{i-1} thread buffers. The merging starts from M_1. If M_i is empty, the thread swaps its thread buffer with the empty buffer and returns to the search phase. Otherwise it merges M_i with its thread buffer, clearing M_i, and proceeds to M_{i+1}. If the a thread reaches M_{k+1}, it writes its thread buffer to disk and returns back to work.

Merge. The ranks are stored in sorted order in multiple files on disk. For interleaving the BWTs, we need to merge the files and to scan through the rank array. We can also use multiple threads here. One thread reads the files and performs a *multiway merge* using a priority queue, producing a stream of lexicographic ranks. Another thread consumes the stream and uses it to interleave the BWTs. If the disk is fast enough, we may want to use multiple threads for the multiway merge.

Implementation

We have implemented the improved merging algorithm as a tool for merging the BWTs of large read collections. The tool, BWT-merge, is written in C++, and the source code is available on GitHub.[2] The implementation uses the *SDSL library* [25] and the new features in C++11. As a result, it needs a fairly recent C++ compiler to compile. We have successfully built BWT-merge on Linux and OS X using g++.

The target environment of BWT-merge is a *single node* of a *computer cluster*. The system should have tens of CPU cores, hundreds of gigabytes of memory, and hundreds of gigabytes of local disk space for temporary files. The number of search threads is equal to the number of CPU cores, while the merge phase uses just one producer thread and one consumer thread. BWT-merge can be adapted to many other environments by adjusting the number and the size of the buffers.

The internal alphabet of BWT-merge is `012345`, which corresponds to either `$ACGTN` or `$ACGNT`, depending on where the BWTs come from. BWTs using different alphabetic orders cannot be merged. We use simple byte-level codes for *run-length encoding* the BWTs. The encoding of run (c, ℓ), where c is the character value and ℓ is the length, depends on the length of the run. If $\ell \leq 41$, the run is encoded in

[2] `https://github.com/jltsiren/bwt-merge`

Table 1: Datasets. The amount of sequence data, the number of reads, and the size of the BWT in the native format and in the Read Server format. RLO indicates that the reads are sorted in reverse lexicographic order. The numbers in parentheses are estimates.

	Data		Native BWT		Read Server	
Dataset	**Size**	**Reads**	**Unsorted**	**RLO**	**BWT**	**FMI**
CEU: All	771 Gbp	7.63G	136 GB	65.9 GB	–	–
Merged	771 Gbp	7.63G	129 GB	58.9 GB	–	–
RS: AA, TT, AT, TA	1.49 Tbp	16.2G	–	136 GB	140 GB	170 GB
Merged	1.49 Tbp	16.2G	–	117 GB	126 GB	(152 GB)
RS: *A, *C	2.45 Tbp	26.5G	–	225 GB	232 GB	281 GB
Merged	2.45 Tbp	26.5G	–	181 GB	197 GB	(239 GB)
RS: *G, *T	2.44 Tbp	26.5G	–	226 GB	232 GB	281 GB
Merged	2.44 Tbp	26.5G	–	180 GB	197 GB	(238 GB)

a single byte as $6 \cdot (\ell - 1) + c$. Longer runs start with byte $6 \cdot 41 + c$, followed by the encoding of $\ell - 42$. The remaining run length is encoded as a sequence of bytes, with the low 7 bits containing data and the high bit telling whether the encoding continues in the next byte. The compressed buffers use the same 7+1-bit code for both the differentially encoded run heads and the run lengths.

For rank/select support, we divide the BWTs into 64-byte blocks of compressed data, ensuring that the runs do not cross block boundaries. For each block i, we store the total number of characters in blocks 1 to $i - 1$ as n_i, as well as the cumulative character counts $c_i = \mathsf{BWT.rank}(n_i, c)$ for $0 \leq c \leq 5$. These increasing sequences are stored using the *sdarray* encoding [26]. To compute $\mathsf{BWT.rank}(j, c)$, we start with a rank query on the n_i sequence to find the block. A select query on the same sequence transforms j into a block offset, while a select query on the c_i sequence gives the rank up to the beginning of the block. We then decompress the block to answer the query. select queries and accessing the BWT work in a similar way. There are also optimizations for e.g. computing $\mathsf{rank}(i, c)$ for all characters c, and for finding the children of a reverse trie node corresponding to a short lexicographic range.

We use two-level arrays with 8-megabyte blocks to store the BWTs and the compressed buffers, managing the blocks using `mmap()` and `munmap()`. This reduces the space overhead by tens of gigabytes over using `malloc()` and `free()`.

Experiments

We used a system with two 16-core AMD Opteron 6378 processors and 256 gigabytes of memory. The system was running Ubuntu 12.04 on Linux kernel 3.2.0. We used a development version of BWT-merge equivalent to v0.3, and the versions of the other tools that were available on GitHub in October 2015. All software was compiled with gcc/g++ version 4.9.2. We stored the input/output files on a distributed Lustre file system and used a local 0.5 TB disk for temporary files. BWT-merge used 32 threads, while the other BWT construction tools were limited by design to 4 or 5 threads.

Our datasets come from phase 3 of the *1000 Genomes Project* [1]. CEU contains 101 bp reads from high-coverage sequencing of the *CEU trio* (individuals NA12878, NA12891, and NA12892). We downloaded the gzipped FASTQ files (run accessions

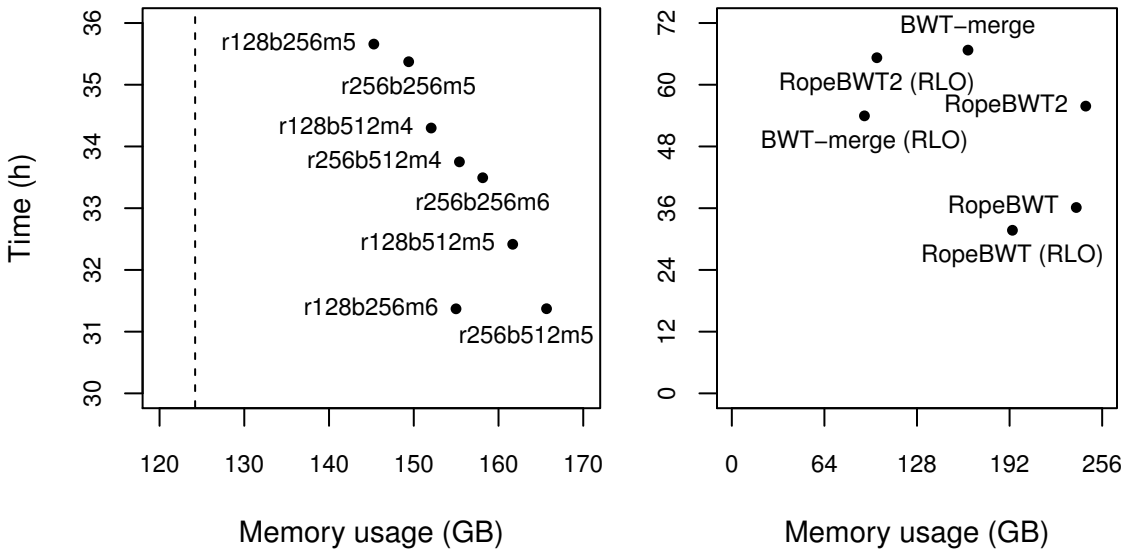

Figure 2: Time/space trade-offs. Left: Merging four BWT files (1.49 Tbp). Label rXbYmZ denotes X MB run buffers, Y MB threads buffers, and Z merge buffers. The dashed line marks the total size of the last pair of BWTs to be merged. Right: Building the BWT of the 771 Gbp CEU dataset. RLO indicates reverse lexicographic order.

SRR622457, SRR622458, and SRR622459). For each individual, we concatenated the files and corrected the sequencing errors with BFC [27] (`bfc -s 3g -t 16`). RS is from the *Read Server* project, which uses all low-coverage and exome data from the phase 3. There are 53.0 billion unique reads for a total of 4.88 Tbp. The reads are in 16 run-length encoded BWTs built by using the *String Graph Assembler* (SGA) [28], partitioned by the last two bases. See Table 1 for further details on the datasets.

Parameters. For testing different parameter values, we took four BWT files (AA, TT, AT, and TA) containing a total of 1.49 Tbp from the RS dataset, and converted them to the *native format* of BWT-merge. This format includes the rank/select structures required by the FM-index. We then merged the BWTs (in the given order). We used 128 MB or 256 MB run buffers and 256 MB or 512 MB thread buffers. The number of merge buffers was chosen so that the files on disk were always merged from either 8 GB or 16 GB of thread buffers.

The results can be seen in Figure 2 (left). The average speed for inserting 1.06 Tbp into file AA ranged from 8.27 Mbp/s to 9.40 Mbp/s, depending on the parameter values. Memory overhead was 21.1 GB to 41.5 GB on top of the 124.2 GB required by the last pair of BWTs. For the further experiments, we chose 128 MB run buffers, 256 MB thread buffers, and 6 merge buffers (overhead 30.8 GB).

Comparison. In the next experiment, we compared BWT-merge to the fastest BWT construction tools on general hardware [18]. We built the BWT of the CEU dataset using RopeBWT [29] with parameters `-btORf -abcr` and RopeBWT2 [18] with parameters `-bRm10g`. We also built individual BWTs using RopeBWT and merged them with BWT-merge. All tools were set to write the BWTs in their preferred formats.

The results are in Figure 2 (right). When the reads are in the original order,

BWT-merge is 1.85x slower and 1.46x more space-efficient than RopeBWT. RopeBWT2 ran out of memory just before finishing. It would have been about 1.2x faster and 1.5x less space-efficient than BWT-merge. The running time of BWT-merge was split evenly between BWT construction and merging. When RopeBWT and RopeBWT2 sort the reads in *reverse lexicographic order* (RLO) to improve compression, all tools improve their performance. BWT-merge becomes 1.70x slower and 2.12x more space-efficient than RopeBWT, and 1.21x *faster* and 1.09x more space-efficient than RopeBWT2.

Read Server. In the last experiment, we merged the 16 BWT files in the RS dataset into two files (AA, CA, TA, GA, AC, CC, GC, and TC into the first file; TT, GT, CT, AT, TG, GG, CG, AG into the second one). Merging the BWTs took 81.3 hours and 83.0 hours, required 221 GB and 219 GB of memory, and used 297 GB and 300 GB of disk space, respectively. This reduced the size of the FM-indexes from around 560 GB to 480 GB. By converting the BWTs to the native format of BWT-merge, we further reduced the size of the indexes to 360 GB.

Conclusions

We have proposed an improved BWT merging algorithm for large read collections. Our implementation of the algorithm in the BWT-merge tool is fast enough to be used with terabases of sequence data. It requires only 30 gigabytes of memory on top of the BWTs to be merged. As BWT-based indexes access large arrays in a random fashion, they must reside in memory in most applications. Hence BWT-merge can build the index on the same system as it is going to be used.

BWT-merge can be used as a part of a BWT construction algorithm. We split the read collection into subcollections, build the BWTs of the subcollections, and merge the results. The resulting algorithm is typically slower but more space-efficient than the existing algorithms.

The most important feature of our algorithm is its low memory usage. With it, we can build the BWTs of much larger read collections than before on commonly available hardware. As a concrete example, we merged the 16 Read Server BWT files into two files. This reduced the number of servers required to host the indexes from three to two, and also improved the query performance of the servers.

In the future, we are going to extend BWT-merge to support different *text orders*, and to optionally *remove duplicate texts* from the merged collection. The current algorithm maintains the existing order by inserting the texts from $\mathsf{BWT}_\mathcal{B}$ after the texts in $\mathsf{BWT}_\mathcal{A}$. This makes it easy to determine the original text identifiers without having to store a permutation. Other text orders are useful for different purposes.

References

[1] The 1000 Genomes Project Consortium, "A global reference for human genetic variation," *Nature*, vol. 526, pp. 68–64, 2015.

[2] D. M. Church *et al.*, "Extending reference assembly models," *Genome Biology*, vol. 16, p. 13, 2015.

[3] V. Mäkinen *et al.*, "Storage and retrieval of highly repetitive sequence collections," *Journal of Computational Biology*, vol. 17, no. 3, pp. 281–308, 2010.

[4] E. Ohlebusch, *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.
[5] V. Mäkinen *et al.*, *Genome-Scale Algorithm Design*. Cambridge University Press, 2015.
[6] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.
[7] Y. Mori. (2008) libdivsufsort. [Online]. Available: https://github.com/y-256/libdivsufsort
[8] G. Nong *et al.*, "Two efficient algorithms for linear time suffix array construction," *IEEE Transactions on Computers*, vol. 60, no. 10, pp. 1471–1484, 2011.
[9] G. H. Gonnet *et al.*, "New indices for text: PAT trees and PAT arrays," in *Information retrieval: data structures and algorithms*. Prentice-Hall, 1992, pp. 66–82.
[10] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Parallel external memory suffix sorting," in *Proc. CPM 2015*. Springer, 2015, pp. 329–342.
[11] W.-K. Hon *et al.*, "A space and time efficient algorithm for constructing compressed suffix arrays," *Algorithmica*, vol. 48, no. 1, pp. 23–36, 2007.
[12] J. Kärkkäinen, "Fast BWT in small space by blockwise suffix sorting," *Theoretical Computer Science*, vol. 387, no. 3, pp. 249–257, 2007.
[13] J. Sirén, "Compressed suffix arrays for massive data," in *Proc. SPIRE 2009*. Springer, 2009, pp. 63–74.
[14] D. Okanohara and K. Sadakane, "A linear-time Burrows-Wheeler transform using induced sorting," in *Proc. SPIRE 2009*. Springer, 2009, pp. 90–101.
[15] P. Ferragina, T. Gagie, and G. Manzini, "Lightweight data indexing and compression in external memory," *Algorithmica*, vol. 63, no. 3, pp. 707–730, 2012.
[16] T. Beller, M. Zwerger, S. Gog, and E. Ohlebusch, "Space-efficient construction of the Burrows-Wheeler transform," in *Proc. SPIRE 2013*. Springer, 2013, pp. 5–16.
[17] M. J. Bauer *et al.*, "Lightweight algorithms for constructing and inverting the BWT of string collections," *Theoretical Computer Science*, vol. 483, pp. 134–148, 2013.
[18] H. Li, "Fast construction of FM-index for long sequence reads," *Bioinformatics*, vol. 30, no. 22, pp. 3274–3275, 2014. [Online]. Available: https://github.com/lh3/ropebwt2
[19] C.-M. Liu, R. Luo, and T.-W. Lam, "GPU-accelerated BWT construction for large collection of short reads," 2014, arXiv:1401.7457.
[20] J. Pantaleoni, "A massively parallel algorithm for constructing the BWT of large string sets," 2014, arXiv:1410.0562.
[21] H. Wang *et al.*, "BWTCP: A parallel method for constructing BWT in large collection of genomic reads," in *Proc. ISC 2015*. Springer, 2015, pp. 171–178.
[22] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.
[23] M. Burrows and D. J. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994.
[24] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane, "Compressed indexes for dynamic text collections," *ACM Transactions on Algorithms*, vol. 3, no. 2, p. 21, 2007.
[25] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Proc. SEA 2014*. Springer, 2014, pp. 326–337.
[26] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *Proc. ALENEX 2007*. SIAM, 2007, pp. 60–70.
[27] H. Li, "Correcting Illumina sequencing errors for human data," 2015, arXiv:1502.03744.
[28] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome Research*, vol. 22, no. 3, pp. 549–556, 2012.
[29] H. Li. (2011-2013) RopeBWT. [Online]. Available: https://github.com/lh3/ropebwt