BIOINFORMATICS ORIGINAL PAPER

Vol. 30 no. 19 2014, pages 2796–2801
doi:10.1093/bioinformatics/btu387

Data and text mining

Advance Access publication June 20, 2014

BEETL-fastq: a searchable compressed archive for DNA reads

Lilian Janin, Ole Schulz-Trieglaff and Anthony J. Cox*

Computational Biology Group, Illumina Cambridge Ltd., Little Chesterford, Essex CB10 1XL, UK

Associate Editor: Jonathan Wren

ABSTRACT

Motivation: FASTQ is a standard file format for DNA sequencing data, which stores both nucleotides and quality scores. A typical sequencing study can easily generate hundreds of gigabytes of FASTQ files, while public archives such as ENA and NCBI and large international collaborations such as the Cancer Genome Atlas can accumulate many terabytes of data in this format. Compression tools such as gzip are often used to reduce the storage burden but have the disadvantage that the data must be decompressed before they can be used.

Here, we present BEETL-fastq, a tool that not only compresses FASTQ-formatted DNA reads more compactly than gzip but also permits rapid search for *k*-mer queries within the archived sequences. Importantly, the full FASTQ record of each matching read or read pair is returned, allowing the search results to be piped directly to any of the many standard tools that accept FASTQ data as input.

Results: We show that 6.6 terabytes of human reads in FASTQ format can be transformed into 1.7 terabytes of indexed files, from where we can search for 1, 10, 100, 1000 and a million of 30-mers in 3, 8, 14, 45 and 567 s, respectively, plus 20 ms per output read. Useful applications of the search capability are highlighted, including the genotyping of structural variant breakpoints and '*in silico* pull-down' experiments in which only the reads that cover a region of interest are selectively extracted for the purposes of variant calling or visualization.

Availability and implementation: BEETL-fastq is part of the `BEETL` library, available as a github repository at `github.com/BEETL/BEETL`.

Contact: acox@illumina.com

Received on April 11, 2014; revised on May 26, 2014; accepted on June 9, 2014

1 INTRODUCTION

Much has been written about disruptive changes in DNA sequencing technology over the last decade and the need for compact ways to store the vast datasets that these new technologies have facilitated. The raw reads in a sequencing project are commonly stored in an ASCII-based format called FASTQ (Cock *et al.*, 2010), the entry for each read comprising a *read ID* string that holds free-text metadata associated with the read, a string for the sequence itself and a string of *quality scores* that encodes accuracy estimates for each base.

The general purpose compression tool gzip (www.gzip.org, Jean-Loup Gailly and Mark Adler) is free and widely available, and is hence an appealing option for compressing FASTQ files. Many bioinformatics tools can read gzip-compressed data directly via the API provided by the zlib library (www.zlib.net, Jean-Loup Gailly and Mark Adler), which avoids the need to create the uncompressed file as an intermediate, but nevertheless still incurs the computational overhead of decompressing the entire file, which is considerable for the large file sizes we are typically dealing with. For many applications, we would like *random access* to the data without the need to decompress the file in its entirety.

This need was recognized by the authors of the Samtools package (Li *et al.*, 2009), which features two programs, razip (a tool available from `razip.sourceforge.net` appears to have similar aims to the eponymous Samtools program, but seems to be an entirely separate development. We deal exclusively with Samtools' `razip` here) and bgzip, that take a *block-compressed* approach to random access while retaining some degree of backward compatibility with gzip. The data are divided into contiguous blocks that are compressed individually, allowing decompression to commence from any point in the file, while limiting the overhead to the need to decompress the data that precede that point within its block.

Given this functionality, it is simple to index a set of records by some key of interest by sorting them in order of that key, block-compressing them and retaining the mapping from key value to file offset for a subsampling of the records. Several of Samtools' user-level tools work in this way, e.g. tabix and indexed BAM files are both indexed by genomic coordinate, whereas faidx uses the name of the sequence as a key.

However, we wish to search for any substring within the sequences, which is not possible with a key-based indexing strategy. Instead, we use an approach based on the Burrows–Wheeler transform, or BWT (Burrows and Wheeler, 1994). The BWT is a reversible transformation of a string that acts as a *compression booster*, permuting the symbols of the string in a way that tends to enable other text-compression techniques to work more effectively when subsequently applied. When the BWT-transformed string is decompressed again, the reversible nature of the transform allows the original string to be recovered from it.

Although it was originally developed with compression in mind, Ferragina and Manzini (2000) showed that, in combination with some relatively small additional data structures, the (compressed) BWT can act as a *compressed index* that facilitates rapid search within the original string. This concept has been highly influential in bioinformatics, being the means by which BWT-based aligners such as BWA (Li and Durbin, 2009) and Bowtie (Langmead *et al.*, 2009) accelerate searches against a reference genome. The core idea is that exact occurrences of some query within the original string can be found by applying a recursive *backward search* procedure to its BWT.

*To whom correspondence should be addressed.

Having found some exact matches to our query within the reads in this way, we continue the recursion to extend these hits into the entire sequences of the reads that contain them. Once the extension reaches the boundary of a read, a lookup into an additional table allows the original positions of the reads in the FASTQ file to be deduced. This last piece of information enables the quality score and read ID strings to be extracted from razip-compressed files that have been indexed using their ordering within the original FASTQ file as a key.

Specifically, given a query DNA sequence q, our tool can provide, in increasing order of computational overhead:

- the number of occurrences of q in the reads,
- the full sequence of each of the reads that contain q,
- the quality score strings associated with the sequences that contain q,
- the read IDs of the reads whose sequences contain q,
- (For paired-read data) the read IDs, sequences and quality scores of the reads that are paired with the sequences that contain q.

Three potential applications demonstrate the usefulness of this fast k-mer search: an '*in silico* pull-down' experiment in which only the reads that cover a region of interest are selectively extracted for the purpose of variant calling or visualization, a *de novo* assembly of reads from insertion breakpoints and the genotyping of structural variants by means of tracking the k-mers that overlap their breakpoints.

2 METHODS

2.1 Definitions

Given a string S comprising n symbols drawn from some alphabet σ, we mark its end by appending a unique symbol \$ that is lexicographically smaller than any symbol in σ. The *BWT* of S is defined such that the i-th symbol of BWT(S) is the character of S immediately preceding the suffix of S, which is i-th smallest in lexicographic order. The concept of the BWT can be readily generalized to encompass a set of strings $S_1, \ldots, S_m$ if we imagine the strings are terminated with distinct end markers that satisfy $\$_1 < \ldots < \$_m$.

The definition of the BWT implies that any occurrence of a symbol within BWT(S) has a one-to-one relationship with a suffix of S that we call its *associated suffix*. Given some query string Q that occurs at least once in S, the characters in BWT(S) whose associated suffixes start with Q form a contiguous subsequence that we call the Q-*interval* of BWT(S). If we have located the Q-interval in BWT(S), the *backward search* procedure allows the position and size of the pQ-interval to be deduced for any symbol p by means of rank() computations, which count the number of occurrences of p within intervals of BWT(S) (see, for example, Adjeroh *et al.*, 2008).

2.2 Index construction

FASTQ data are first split into its three component streams: bases, read IDs and quality scores. The read IDs and quality scores are each dealt with in the same way: compressed with razip and augmented with an index that—for every 1024th read—stores the offset in the file at which the data associated with that read begin. The BWT of the sequences is built using the algorithm described in Bauer *et al.* (2011, 2013): we use our own BEETL library for this, but we also note the several additional improvements in Heng Li's implementation (`github.com/lh3/ropebwt`) and promising recent work by Liu *et al.* (2014) that demonstrated accelerated BWT construction using GPU technology.

During BWT construction, we also generate an 'end-pos' file containing an array that maps between the ordering of the read associated with each \$ sign and its read number in the original FASTQ file. The BWT itself is stored in a manner similar to that used by the sga assembler (Simpson and Durbin, 2012), runs of characters being represented by byte codes, the least significant bits encoding the character and the remainder of the byte denoting the length of the run. To speed the rank() calculations needed for backward search, we create a simple index of the BWT files by storing, once every 2048 byte codes, the number of times each character has been encountered so far in the BWT.

The index construction flow is shown in the top half of Figure 1, and Table 1 summarizes the files generated by BEETL-fastq to store and index the original FASTQ files.

2.3 Searching for a query sequence in the index

BEETL-fastq's search mode consists of three main computation stages shown in the lower half of Figure 1:

- BEETL-search performs the k-mer search and retrieves the Q-interval matching each k-mer.
- BEETL-extend propagates each BWT position from these Q-intervals to the end of each read, where we are able to identify the read number. At the same time, BEETL-extend's propagation of the k-mers reconstructs each base of the reads.
- From the read numbers, we extract read IDs and quality scores using the razip files.

The read IDs, bases and quality scores are then interleaved to generate the output FASTQ file.

3 RESULTS

3.1 The Platinum Genomes dataset

In the following sections, we refer to several datasets from Illumina's Platinum Genome (PG) project (Eberle *et al.*, manuscript in preparation), which aims to systematically identify all variants in a large three-generation family (the CEPH/Utah

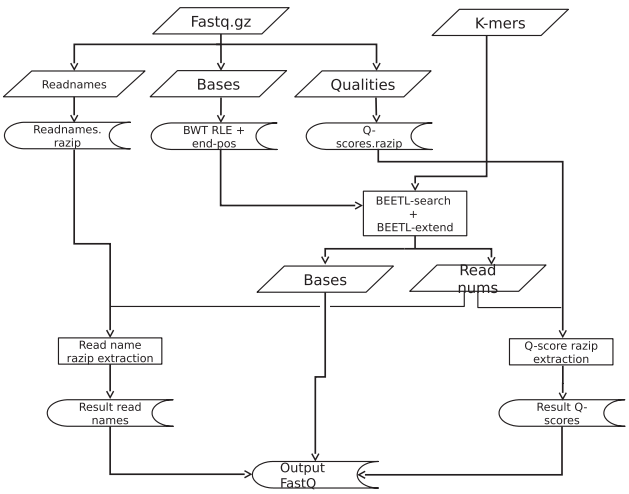

Fig. 1. BEETL-fastq flow

pedigree 1463) by combining multiple variant calling approaches and making use of the inheritance structure within the pedigree. The 17 members of the pedigree were each sequenced to 50-fold coverage, the total dataset comprising 27 billion reads of length 100, requiring 6.6TB of storage in FASTQ format (or 2.5 TB if compressed with gzip). Both the raw sequences and variant calls are publicly available (www.illumina.com/platinumgenomes/).

Our experiments with these genomes involved repeated searches of *k*-mers. Our strategy was to prepare and search each genome independently to get the benefits of distributed processing. One of them, designated NA12877, comprises 165.4 GB of paired 100-mers, stored as gzipped FASTQ files totalling 151.71 GB, equivalent to 390.55 GB of uncompressed FASTQ data. Compressing and indexing these data with BEETL-fastq (i.e. conversion of gzipped FASTQ into BWT plus razip-compressed quality scores and read IDs) took 7.5 h on an Amazon EC2 i2.2xl instance and reduced the data to 113 GB, a reduction of 26%. From this, search query results were obtained at a rate presented in Table 2, while the subsequent read extraction stage consistently took around a further 20 ms per read.

An interesting option would be to store all 17 datasets as one compressed entity. The redundant information present in these genome reads would be grouped together by the BWT permutation, potentially leading to a better overall compression. However, we would lose the 17-way parallelism that allows us to achieve the search speed presented here. As our number of datasets increase, we will group datasets together while still keeping enough concurrency to match the required search time. Keeping the datasets separated also allows us to target specific members of the pedigree independently.

3.2 Comparison of read ordering strategies

Ordering the reads before indexing affects the time taken to build the index and the time taken to build and query it. Here we compare three different strategies:

- *Unordered:* reads are left in the original FASTQ order,
- *Lexicographic order* (*LO*)*:* reads are sorted in lexicographical order of their bases,
- *Reverse lexicographic order* (*RLO*)*:* reads are sorted in lexicographical order of their reversed (not reverse-complemented, although it would be equivalent) string.

Table 3 compares these strategies on the NA12877 reads. Compression and search times were measured on a server with 16 2.5 GHz cores, although BEETL-fastq used at most six cores. BWT construction is I/O-bound, and the underlying RAID disk was averaging 100 MB/s.

The advantages of each strategy can be summarized as follows:

- *Unordered:* read IDs compress better, as they are usually generated in an ordered way, which gets shuffled by the other strategies. This is a small gain.
- *LO:* the dataset-end-pos file is not needed, as the '$' signs in the BWT end up in the same order as the original reads. This saves 7% of the total size.
- *RLO:* the BWT files get longer runs of identical letters, leading to a better compression rate (45% of BWT size, 7% of total size) and to faster search time (20% faster).

The RLO sort achieves a 7% reduction of the total size compared with the unordered strategy. This comes at an extra cost in initialization time, a stage where optimizations are still possible. However, as our flow needs a single initialization for an unlimited number of searches, the main benefits come at search time, where the I/O-bound search needs to go through a BWT, which is 45% smaller. Experiments confirm a 45% faster search.

In the case of a server accessing files remotely, as described in the next section, another advantage lies in the fact that only BWT index files are kept cached in RAM. Having a BWT 45% smaller means that a 45% smaller RAM footprint is achieved with RLO.

It should be noted that the search-only time reported here corresponds to the first stage of reads extraction. The second stage (extension of BWT positions) is also accelerated by 45%, as it is based on the same BWT suffix extension algorithm, whereas the third stage (extraction of quality scores and read Ids from razip files) is unaffected.

3.3 Cloud-based deployment

One interesting potential application is as a search service, answering the queries of users on demand and saving them from the need to download large FASTQ files in full.

A range of use cases was explored, from very fast search of a few *k*-mers to longer batch search of millions of *k*-mers. Here we report on the two extreme cases: large queries of millions of *k*-mers under 1 h, and queries of a single *k*-mer expected to return results under 1 s, this last use case leading to a useful cloud

Table 1. Files generated by the indexing process

File name	Description
dataset-B0[0-6]	BWT files in run-length encoded format
dataset-B0[0-6].idx	BWT index files for faster random accesses
dataset-end-pos	Read numbers of the '$' signs in BWT
dataset.quals.rz	razip-compressed quality scores, able to start decompressing at any given position
dataset.quals.rz.idx	index file from compressed quality scores, to map read numbers to character positions in the razip file
dataset.readIds.rz	razip-compressed read IDs
dataset.readIds.rz.idx	index file for compressed read IDs

Note: Together, these files comprise a lossless representation of the data in the original FASTQ files.

Table 2. BEETL-fastq search speed from NA12877 (time in seconds)

Number of 30-mers	1	10	100	1000	1 000 000
BEETL-search (30 cycles)	1	1.8	2.9	7.2	207
BEETL-extend (100 cycles)	1.4	6.1	11.2	37.5	360

configuration. Those timings are achieved on an Amazon EC2 instance querying indexes of the set of 17 human genome datasets described in Section 3.1, the indexes being held in Amazon S3 storage.

Both use cases answer the following constraints in different manners:

- how to efficiently access the datasets stored on S3,
- what level of parallelization is needed,
- how to minimize the costs.

Keeping instances running helps achieve good speed, as the data can be kept local, even in RAM if necessary. But this usually comes at a high cost. On the other hand, spawning Amazon instances on demand takes a few minutes just to start the service.

3.3.1 Large batch queries Large searches of millions of *k*-mers are actually simple, as the time needed to spawn a large Amazon instance and download the data are small enough compared with the computation time.

Running a cluster of 17 Amazon instances (the number of genomes in our dataset) with 120 GB of SSD and 30 GB of RAM such as m3.2xlarge satisfies our constraints. The instances achieve good overall download speed from S3, each being able to keep one full genome files on disk and its BWT in RAM. Keeping the genome searches on separate instances prevents RAM contention.

3.3.2 Small queries with fast response Ideally, we would like a search for one *k*-mer within our dataset to yield a response within 1 s. However, in practice this timing greatly depends on the number of matching reads as the final razip extraction takes around 20 ms per read. We decided to distinguish between the three phases of a search: *k*-mer search in BWT, extension of BWT positions to bases and read numbers and extraction of qualities and read IDs. We focus here on the first stage, the *k*-mer search, which returns the number of matching reads. This is useful information by itself, for example in genotyping applications.

The first challenge was to decide where the tools can run: an Amazon instance takes at least 1 min to start up, and downloading one of the human genome datasets from S3 to an instance takes, in the best case, 3 min for the sequences alone (around 16 GB, assuming 100 MB/s). One solution is to prefetch the BWT files local to an always-on Amazon instance. However, this is an expensive solution if the Amazon instance is large enough to process 17 human genome datasets in parallel.

A cheaper alternative is to leave the BWT files in Amazon S3 storage and access them remotely, transparently mounted as files using httpfs (`sourceforge.net/projects/httpfs`). This is a high-latency solution, which ends up working well. The index files are pre-fetched and kept in RAM, and the quantity of RAM needed is low enough that we can now pick a much cheaper instance (such as m3.medium).

The organization of our file system is such that the files described in Table 1 appear in one working directory. Large files accessed via httpfs are mounted in subdirectories and symbolic links make them appear correctly in the working directory. Small index files are directly stored in the working directory. With the default BEETL-search tool, search queries of a single 30-mer were at this stage answered within 3 s.

An extra optimization was necessary to bring this down to 1 s; instead of loading the index files from disk to RAM at the start of each BEETL-search, a shared memory structure keeps the index data permanently in RAM, with the ability to be re-used across successive BEETL-search processes.

To make these results easily reproducible and for easy discovery of our tools architecture, a public Amazon AMI image was created, containing all the tools described here: BEETL tools, prefetched index files and httpfs remote mounts of the BWT files for the NA12877 human genome sample from S3. The image also includes a web server able to launch *k*-mer queries on NA12877. This image appears as 'BEETL-fastq Bioinformatics' in Amazon AMI (Amazon Machine Image) search.

Table 3. Effect of sorting strategies on size of the components of the index [in GBytes (bits per base)], build time and search time for NA12877

	Unsorted	LO sort	RLO sort
*NA12877-B0**	33 (1.71)	29.4 (1.53)	18 (0.93)
NA12877-end-pos	7.6 (0.39)	0 (0)	7.6 (0.39)
*NA12877.qual.rz**	64.5 (3.35)	64 (3.32)	64.1 (3.33)
*NA12877.readIds.rz**	7.9 (0.41)	14.9 (0.77)	14.9 (0.77)
Total:	113 (5.87)	108.3 (5.62)	104.6 (5.43)
Reduction versus fastq.gz (%)	–26	–29	–32
Compression time (min)	590	602	647
Search-only time for one 30-mer (ms)	505	460	280

4 APPLICATIONS

4.1 An '*in silico* pull-down' experiment

We used BEETL-fastq to simulate a targeted sequencing experiment *in silico* by extracting all read pairs from NA12878 (another member of the PG pedigree) that overlap RBM15B, a single-exon gene of length 6.6 kb. First, a non-overlapping set of 34-mers covering the gene was extracted from the human reference sequence. The use of non-overlapping sequences limits the redundancy of our queries, but BEETL-fastq can deal equally well with overlapping *k*-mers. We chose a length of 34 because previous studies (Li *et al*., 2010) suggest that 80% of the genome can be uniquely targeted by a sequence of this size, but the only restriction on query length is that imposed by the length of the reads themselves.

Tuning the query *k*-mer length allows a trade-off between sensitivity and specificity, and prior knowledge of the region being targeted can guide this decision. Genomic regions of lower expected coverage or with many variants will require shorter and possibly overlapping *k*-mers, whereas longer *k*-mers will better target more repetitive regions. One could also imagine performing several queries over a range of *k*-mer lengths and then merging the retrieved read sets. This will obviously result in an increased running time but will also increase sensitivity.

Next, we used BEETL-fastq to retrieve the FASTQ entries for all read pairs in NA12878 for which at least one read of the pair

contained an occurrence of one of the query *k*-mers. We then aligned the read pairs with bwa-mem and performed variant calling using Samtools (Li *et al*., 2009); 100% of the retrieved read pairs aligned to the human reference genome, 99.93% of them within the target region of RBM15B, illustrating the high specificity of our approach. The RBM15B gene was covered to 47.32× on average, similar to the depth of 47.47 to which the gene is covered after whole-genome alignment of the entire NA12878 read set. We identified a homozygous $T \rightarrow C$ mutation in RBM15B, which was confirmed to be part of the curated list of variants for NA12878 from the Genome in a Bottle project (Zook *et al*., 2014).

The whole process takes only minutes, and the time is dominated by the alignment of the retrieved reads to the whole reference genome. This could be sped up considerably by limiting the search space for the alignment to the subregion being targeted, but, even unoptimized, the BEETL-fastq approach is orders of magnitude faster than whole-genome alignment when we are only interested in a small number of genes and their variants.

To assess sensitivity, we compared the reads mapped with RBM15B during whole-genome alignment to the read set retrieved by BEETL-fastq. BEETL-fastq retrieved 99.97% of the reads mapped by whole-genome alignment to the target region. The reads that were missed by BEETL-fastq include reads mapped with many mismatches, soft-clipping or to edges of the target region.

4.2 *De novo* assembly of insertions

One of the advantages of BEETL-fastq is its ability to extract not only the read containing the query *k*-mer but also its read partner from the same DNA fragment. This opens new applications not accessible to other tools such as *de novo* assembly of breakpoint regions but also dissection of complicated breakpoint regions. Unfortunately, the detection of large insertions is still difficult for the current generation of structural variant calling tools and comprehensive annotations even for well-characterized genomes such as NA12878 is difficult to obtain. To give an example that it is nevertheless possible to fully assemble large insertions using the query results of BEETL-fastq, we are examining an insertion, which we detected using an in-house structural variant calling pipeline as well by manual inspection. The insertion occurs on chromosome 11 of NA12878 at position 5 896 446 and has a total length of 253 bp. We extract the set of tiling 35-mers from the insertion as well as the *k*-mers crossing the insertion breakpoints.

Using BEETL-fastq, we extract all read pairs in which at least one read contains an insertion *k*-mer. We then assemble these read pairs using the velvet assembler (Zerbino and Birney, 2008) using standard parameters and a *k*-mer length of 31. The result is a single contig, which we successfully aligned back to the reference genome and could confirm the insertion site.

4.3 Structural variant breakpoints

The genome NA12878 from the PG pedigree was also sequenced as part of the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2012), which compiled a list of 2702 deletions in NA12878 from the output of multiple SV callers (Mills *et al*., 2011). After removing calls annotated as being of low quality or having imprecise breakpoints, we were left with 1209 high-confidence deletion calls with precise breakpoints and estimated genotypes.

We extracted the set of 34-mers crossing the deletion breakpoints and their reverse complements. We queried the FASTQ files of NA12878 for these *k*-mers to confirm the annotation by the 1000 Genomes consortium. We could retrieve reads for 1067 out of 1209 deletions (88.3%). We inspected some of the breakpoints for which we could not extract any *k*-mers and found these breakpoints either have ambiguous locations or are closely adjacent to SNPs or small indels.

As a next step, we compute the read coverage across the breakpoint from the reads extracted by BEETL-fastq and compared the results to the genotypes computed by the 1000 Genomes consortium. For heterozygous events, the coverage should be approximately half of the whole-genome coverage, whereas for homozygous events breakpoint and whole-genome coverage should be approximately the same. We found the breakpoint coverages to be close to the expected values, a mean value of 17.2 for heterozygous and 29.52 for homozygous deletions. Figure 2 shows the spread of read coverages. There are some outliers which occur very frequently in the data. Closer inspections reveal that they mostly map to short repeat instances.

Having confirmed that our genotype estimates match, we can test for the existence of these deletions in other samples by querying for the presence of these characteristic *k*-mers in the indexes of their reads. This is a powerful method and allows us to process large numbers of unaligned FASTQ files in a rapid manner. We can also genotype and map variations across families and populations.

As an example, we examine the parents of NA12878, who are also part of the PG pedigree, having the sample IDs NA12891 and NA12892. We query the indexes of their reads to check if the estimated coverage matches the inheritance pattern given the genotypes in NA12878. For instance, if NA12878 contains a homozygous deletion, we would expect to find the corresponding

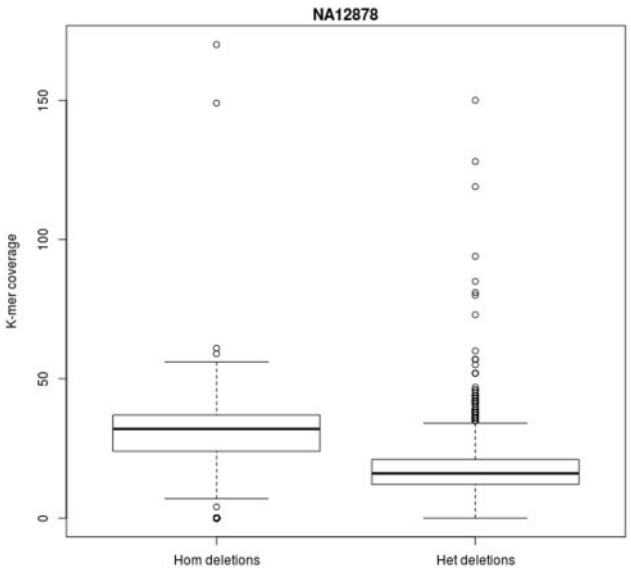

Fig. 2. Read coverage of 1000 Genomes deletions in NA12878

k-mers in both parents. We found that the k-mer occurrences match this expected pattern in 99.96% of all cases.

5 DISCUSSION

Our focus here has been to represent the input data in a lossless fashion, but Table 3 highlights that the relative storage needs of the sequences, quality scores and read IDs perhaps do not reflect their relative importance, as the sequences and their indexes take up only 17% of the total size, barely more than the 14% consumed by the read IDs.

There remains some scope for further lossless compression: in earlier work that focused exclusively on compression (Cox *et al*., 2012), we achieved sub-0.5 bpb on the sequences using 7-zip (www.7-zip.org, Igor Pavlov). Here, however, we sacrifice some compression for the faster decompression, and thus faster search that our simple byte code format gives us. The byte coding can be further optimized, and it may also be advantageous to switch between run-length encoding and naive 2-bits-per-base encoding on a per-block basis, choosing whichever of the two strategies best compresses each block.

However, to achieve significant further compression, some degree of lossy compression is likely to be necessary. Each of the three data streams is potentially amenable to this, and our methods can of course still be applied to the resulting data.

The free format of the read IDs limits our ability to comment generally on the prospects of compressing such data further. Nevertheless, it could be argued that, for paired data, the most useful metadata to retain is which read is paired with which: this could be simply encapsulated in an array containing one pointer per read, consuming $O(\log n)$ bits.

The space taken up by the sequences themselves would be reduced by error correction, two possible strategies being the trimming of low-quality read ends [as demonstrated by Cox *et al*. (2012)] or a k-mer-based approach such as Musket (Liu *et al*., 2013).

Lastly, the majority of the archive's size is taken up by the quality scores. More recent Illumina data default to a reduced-representation quality scoring scheme that makes use of only 8 of the 40 or so possible quality values, but the PG data we tested still follow the full-resolution scoring scheme. The newer scheme would likely reduce the size of the scores by about half. We have also described a complementary approach (Janin *et al*., 2014) that uses the k-mer context adjacent to a given base to decide whether its quality score can be discarded without likely detriment to variant calling accuracy.

Conflicts of Interest: All authors are employees of Illumina Inc., a public company that develops and markets systems for genetic analysis, and receive shares as part of their compensation.

REFERENCES

Adjeroh,D. *et al.* (2008) *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. 1st edn. Springer US.

Bauer,M.J. *et al.* (2011) Lightweight BWT construction for very large string collections. In: *CPM (2011)*. LNCS. Vol. 6661, Springer Berlin Heidelberg, pp 219–231.

Bauer,M.J. *et al.* (2013) Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, **483**, 134–148.

Burrows,M. and Wheeler,D.J. (1994) A block sorting data compression algorithm. In: *Technical report*. DIGITAL System Research Center.

Cock,P.J.A. *et al.* (2010) The sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res.*, **38**, 1767–1771.

Cox,A. *et al.* (2012) Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *Bioinformatics*, **28**, 1415–1419.

Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, pp. 390–398.

Janin,L. *et al.* (2014) Adaptive reference-free compression of sequence quality scores. *Bioinformatics*, **30**, 24–30.

Langmead,B. *et al.* (2009) Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*, **10**, R25.

Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, **25**, 1754–1760.

Li,H. *et al.* (2009) The sequence alignment/map format and samtools. *Bioinformatics*, **25**, 2078–2079.

Li,R. *et al.* (2010) *De novo* assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, **20**, 265–272.

Liu,C.M. *et al.* (2014) GPU-accelerated BWT construction for large collection of short reads. *arXiv: 1401.7457*.

Liu,Y. *et al.* (2013) Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, **29**, 308–315.

Mills,R.E. *et al.* (2011) Mapping copy number variation by population-scale genome sequencing. *Nature*, **470**, 59–65.

Simpson,J.T. and Durbin,R. (2012) Efficient *de novo* assembly of large genomes using compressed data structures. *Genome Res.*, **22**, 549–556.

The 1000 Genomes Project Consortium. (2012) An integrated map of genetic variation from 1,092 human genomes. *Nature*, **491**, 56–65.

Zerbino,D.R. and Birney,E. (2008) Velvet: algorithms for *de novo* short read assembly using de bruijn graphs. *Genome Res.*, **18**, 821–829.

Zook,J.M. *et al.* (2014) Integrating human sequence data sets provides a resource of benchmark SNP and indel genotype calls. *Nat. Biotech.*, **32**, 246–251.