Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

OXFORD

Genome analysis

CSAM: Compressed SAM Format

Rodrigo Cánovas [1,2,*], Alistair Moffat [2] and Andrew Turpin [2]

[1] L.I.R.M.M. & Institut Biologie Computationnelle, Université de Montpellier, CNRS F-34392 Montpellier Cedex 5, France.
[2] Department of Computing and Information Systems, The University of Melbourne, Victoria 3010, Australia.

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Next generation sequencing machines produce vast amounts of genomic data. For the data to be useful, it is essential that it can be stored and manipulated efficiently. This work responds to the combined challenge of compressing genomic data, while providing fast access to regions of interest, without necessitating decompression of whole files.
Results: We describe CSAM (Compressed SAM format), a compression approach offering lossless and lossy compression for SAM files. The structures and techniques proposed are suitable for representing SAM files, as well as supporting fast access to the compressed information. They generate more compact lossless representations than BAM, which is currently the preferred lossless compressed SAM-equivalent format; and are self-contained, that is, they do not depend on any external resources to compress or decompress SAM files.
Availability: An implementation is available at `https://github.com/rcanovas/libCSAM`.
Contact: canovas-ba@lirmm.fr
Supplementary Information: Supplementary data is available at *Bioinformatics* online.

1 Introduction

Current next-generation sequencing technologies produce millions of small DNA fragments (*reads*) at once (Church 2006; Mardis 2008; Ansorge 2009; Myllykangas *et al.* 2012), generating file sizes in the gigabyte range at a cost of just a few hundred dollars. Each generated read is a continuous fragment of data extracted from the processing of a single genome, stored as a string of *bases*. In this paper we consider reads composed of four fundamental bases A, C, G, and T, with the inclusion of the letter N, which is used to symbolize bases that could take any value. A number of meta-data fields are associated with each read to form *alignment read* information. Some of these fields add considerably to the space requirement; in particular, the Quality field (QUAL), which measures how accurate the bases of the read are with respect to a reference genome, typically requires (uncompressed) the same space as the sequence of bases (Ewing and Green 1998; Ewing *et al.* 1998; Richterich 1998).

Several standard formats for storing alignment reads have been adopted, each aiming to make it easy to parse and then manipulate them using text-processing tools. The most common representations are the FastA, FastQ (Cock *et al.* 2010), and SAM, or Sequence Alignment Map (Li *et al.* 2009) approaches. Of these, SAM is dominant, partly because it includes more information about each alignment than the other formats. SAM has become one of the most used formats for storing alignment data, in no small part because it is the output generated by many aligners[1]. For example, the compressed version of the SAM format, BAM (Section 2), is currently the preferred structure of the 1000 *Genome Project*[2].

In this work, different approaches that compress SAM files are explored. Most of the techniques described focus on methods that compress reads and/or their associated QUAL fields, which, as we will describe, are the fields that dominate the space requirement of compressed SAM files. Another reason for focusing on these two fields is that most of the remaining fields can be derived from these two. We also consider the problem of random access into the stored data, providing data structures that allow the extraction of segments of the information stored without the need to decompress the whole compressed file.

Finally we introduce a new compressed SAM format, CSAM, which uses less space for storing the data than the BAM format (taking similar or lower times to compress, decompress and access the data), and also supporting queries over multiple alignments without requiring whole files to be decompressed. The CSAM format compresses the data without using

[1] `http://samtools.sourceforge.net/swlist.shtml`, March 2016
[2] `http://www.1000genomes.org/`, March 2016

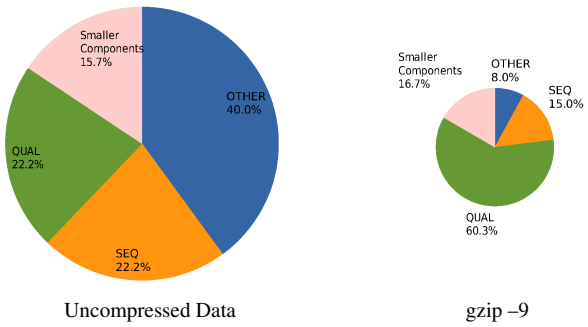

Fig. 1. Space percentages used for the components of a typical SAM file (`NA12878.chrom20.ILLUMINA.bwa.CEU.low_coverage.20120522.sam`, see Table 2), when separated into different components and compressed on a per-component basis using gzip –9 (maximum compression). All percentages are relative to the total space of the respective file, with the components that use less than 10% of the total space grouped under the label of "Smaller Components". In this case, the gzip version compressed the file components to a total of approximately 18% of the original size. As can be seen, the components SEQ, QUAL, and OTHER dominate the storage in both the original and compressed representation.

any external extra information and is currently the only lossless SAM compressor, beside BAM itself, that offers a full compression technique supporting random access to the data. Furthermore, CSAM is the first SAM lossy compression approach allowing random access to the stored data. We also explore how the proposed compression techniques affect the performance of possible uses of the compressed data.

1.1 SAM format

SAM-format data is stored as multi-line, TAB-delimited, plain text that contain two kinds of lines: header lines; and alignment lines. The header lines are optional, and contain commentary, information about the reference sequence used for the alignment, the program used to generate the alignments, and so on. Each alignment line contains 11 or more fields, where the first 11 must be present, but might be "`*`" or "`0`", meaning that the information is not supplied. The order in which these fields appear in each line is always the same. Table 1 in the supplementary material, provides an overview; the SAM project web page[3] provides more details of SAM format and its fields.

To show the proportion of the space used by the various SAM components we extracted each of them into separate files, and measured their raw size and also their gzip-compressed sizes, the latter to provide a sense of how repetitive and compressible each type of data is. Figure 1 shows the results of this before and after measurement. The bulk of the space consumption of the original SAM file arises in the fields SEQ, QUAL, and OTHER, with the QUAL field an even larger fraction of the compressed space. Note that OTHER is one of the optional fields, and while it is dominant in this file, may not be present in other SAM files.

1.2 Compression

We categorize compression modalities into three distinct classes: *lossless*, *information-preserving*, and *lossy*. Each class has advantages and disadvantages, depending on the importance of the information that is stored, and on the eventual use of the data.

Lossless, or exact compression, ensures that the decompressed data is exactly the same as the original, and means that the compressed version must contain sufficient information for the decompresser to generate a bit stream identical to the one that was input to the compressor. *Lossy* compression modalities store an approximate representation of the input data, trading loss in fidelity of reproduction for enhanced compression effectiveness. Sitting between these two modalities, *information-preserving* compression systems guarantee that all the information of the original file is stored, but the order in which it is regenerated might be different than in the original file. In the case of SAM compression, it is assumed that the order in which the reads are processed does not have any meaning, and thus it is possible to reorder the reads without losing information. In particular, the approaches described in this paper are lossless if the input file is already ordered by sequence reference name and alignment start position, and are information-preserving if the input file must be sorted into that order before being processed (for example, using SamTools). Note further, that all of the schemes in this paper that allow random access to compressed data, including our own, assume that the input SAM file is ordered by reference name and relative position.

1.3 Random Access Operations

When large amounts of data are compressed and stored, it is desirable to be able later to extract information without needing to decompress the entire data set. In other words, it is beneficial to support random access operations that extract only the part of the data that is of interest. To achieve this goal, auxiliary information about how the data is stored is required. For example, the BAM format generates an extra index to allow queries over the compressed data, supporting two basic queries:

- *getInterval*(*rname*), which returns the set of data lines with reads that were aligned against the reference *rname*; and
- *getInterval*(*rname, x, y*), which returns the set of data with reads that were aligned against the reference *rname*, with alignment positions in the interval $[x, y]$.

In Section 3 we discuss how we support these two queries in the compressed format proposed, and support for further operations.

1.4 Downstream Applications

An important process in the analysis of genomic data information, is computing the coverage, that is, the number of reads stored that represent a desirable zone; for example exon locations (parts of DNA that are converted into mature messenger RNA) or a specific gene (Lawrence *et al.* 2013; Liao *et al.* 2014; Anders *et al.* 2015). We refer to these zones as *genomic features*. In this work we consider the use of the featureCounts program (Liao *et al.* 2014), which is "a highly efficient general-purpose read summarization program that counts mapped reads for genomic features"[4]. Input consists of one or more SAM/BAM files and a list of genomic features, which can be in *general feature format* (GFF)[5] or *simplified annotation format* (SAF)[6], and the number of reads assigned to each feature is output, plus statistical information for the overall summarization results.

Section 4.4 explores the featureCounts downstream application, showing how our compression approach can be included as input to the program, and measuring the time required, comparing against the times obtained using SAM or BAM input files.

[3] `http://samtools.sourceforge.net/`, March 2016

[4] `http://bioinf.wehi.edu.au/featureCounts/`, March 2016

[5] `http://www.sanger.ac.uk/resources/software/gff/spec.html`, March 2016

[6] `http://bioinf.wehi.edu.au/subread-package/SubreadUsersGuide.pdf`, March 2016

2 Previous Approaches

The BAM format (Li *et al.* 2009) is a binary representation of a corresponding SAM file that uses about 30 per cent of the original space by employing the *BGZF* (Blocked GNU Zip Format) lossless compression suite, which is an augmented form of the standard gzip file format. Each compressed block contains a gzip'ed representation of a span of data, preceded by extra fields that give specific information about the file that has been compressed. In addition, the BAM format (when it is ordered by reference name and alignment start positions) offers the option to store an index containing information about the block's data, permitting fast retrieval of alignments given a specified region, only decompressing and getting the data from the block of interest. For example SamTools[7], allows the query *getInterval*(*rname, x, y*) to be answered without decompressing the whole compressed file.

CRAM (Fritz *et al.* 2011) is another compressed representation of SAM/BAM files, generated using CramTools[8], a suite of Java software and APIs that compress the DNA sequence and quality data. CRAM offers better information-preserving compression than BAM, supporting fast transition between these two formats. A lossy compression mode is also supported, enabling users to choose which data should be preserved. The only dependency of the CRAM format is to an external reference genome. Each sequence is stored as the difference between itself and the external reference, and the same external reference genome must thus be provided each time compression or decompression is undertaken.

In practice the information flow between SAM, BAM and CRAM files is not completely preserved (see supplementary material). To effectively compress a read sequence, CramTools stores only the information of the parts of the read that are identical or near identical to the input reference sequences, otherwise the information is not stored. In addition, CramTools allows several user-defined options that control the lossy compression of the quality score information. Finally, CramTools offers fast transition between CRAM and BAM files, but does not supply random access to the information in CRAM format. Random access queries are supported using SamTools over BAM files, requiring CRAM files to be decompressed first.

In 2013 Popitsch and von Haeseler developed their NGC tool (Popitsch and von Haeseler 2013), offering a mechanism that compresses data stored in SAM/BAM format, covering only reads for which mapped information is available (that is, for which a RNAME field is specified). Popitsch and von Haeseler present a pseudo information-preserving solution, and a lossy solution.

NGC focuses on compressing the read and quality score sequences. To compress the Read Sequences (SEQ, RNAME, POS) field, NGC assumes that the reference sequence is an external fixed input provided by the user at compression and decompression time. NGC stores the differences between the reads and the reference sequence, instead of compressing the reads independently. In order to store the quality scores, the NGC tool offers a simple lossless compression mode in which the quality scores are compressed using bzip2 as a separate stream. If lossy compression is desired, NGC provides a binning strategy based on the LogBinning approach described by Wan *et al.* (2012), which transforms quality scores within predefined intervals to single representative values.

The NGC mechanism does not support random access to the compressed data, and is a storage scheme only.

[7] `http://samtools.github.io/hts-specs/SAMv1.pdf`, March 2016

[8] `http://www.ebi.ac.uk/ena/about/cram_toolkit`, March 2016

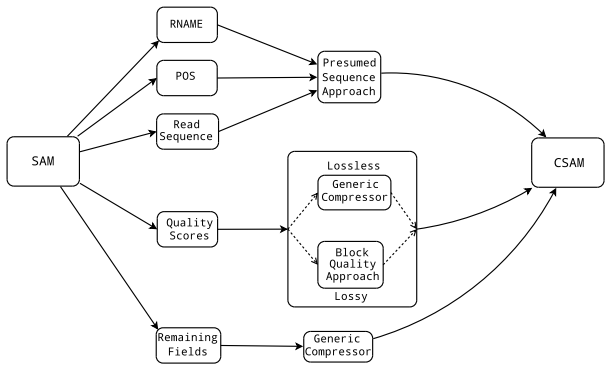

Fig. 2. CSAM compression scheme. The RNAME, POS, and SEQ fields are compressed using the presumed sequence approach, and the QUAL field is losslessly represented using gzip or lossily compressed using the block quality approach presented in by Cánovas et al. (2014). The remaining fields are separately compressed with gzip.

3 Methods

In the new CSAM format we focus primarily on improved representations for the SEQ and QUAL fields, since those two still occupy significant amount of space after compression. Note that the OTHER field is usually highly repetitive and compresses well with gzip. Figure 2 illustrates the compression scheme followed by CSAM, where the fields RNAME, POS, and SEQ are compressed together using the "presumed sequence" approach described by Cánovas and Moffat (2013) (an overview is provided in the supplementary material); the QUAL field is losslessly compressed using gzip or, if lossy modality is desired, using a block quality compression approach such as P-Block or R-Block, described by Cánovas *et al.* (2014); and each of the remaining fields are separately compressed using gzip, assuming that any additional gain in space derived through the use of another lossless technique would be of only marginal overall benefit. When we say that a field is compressed using gzip, we mean that the information is divided into blocks of λ data lines, and then each block is compressed separately with gzip. This allows random access to the stored information. Other general-purpose compression tools might also be used, each offering a different balance between processing cost and compression effectiveness. We opt to use gzip in part because it represents an excellent compromise between these, and in part to be fair when comparing against BAM, which also uses gzip.

We also consider the problem of random accessing the compressed data, in particular, the *getInterval* operation described in Section 1.3. To that end, we add an auxiliary index containing synchronization points within the compressed data to allow extraction of segments starting from these those points. Two different criteria for choosing the synchronization points were explored: inserting them after some fixed number of data lines, or inserting them after some fixed number of base positions. The first of these approaches requires storing an index into the compressed data indicating the decoding start position for every λ th encoded data line. That is, the interval between two consecutive synchronization points spans λ data lines. The second criteria requires storing a pointer to the first data line encoded after every ρ th base position. We refer to these two options as *block sample* and *position sample* respectively. The BAM format provides another possible criteria which stores synchronization points every time a certain number of input bytes has been compressed.

In order to support the operation *getInterval*(*rname, x, y*), where *rname* is a reference name, and $[x, y]$ defines an interval to be searched, CSAM also stores extra information about the data contained between synchronization points, depending on the method used to generate the index. For example, if a block sample or the BAM approach is used

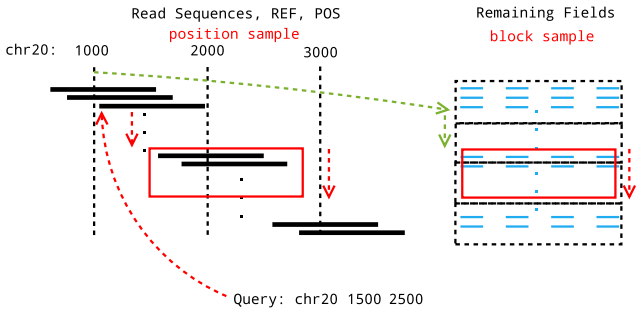

Fig. 3. Example showing the query "chr20 1500 2500" (where "chr20" refers to the chromosome 20) is performed using the CSAM position and block sample indexes. The green arrow indicates how the position sample is associated with the block sample, the red arrows show the path following to extract the interval, and the red rectangles are the information desired.

to generate the index, then is necessary to know the positions (POS field value) and reference name (RNAME field value) that correspond to the read associated with each synchronization point. A search over the stored positions and reference names can then be performed to find which synchronization points need to be accessed. Note that in both cases, block samples and BAM indexes, the accuracy of the search is limited by the parameter chosen to create the index, meaning that the amount of decoding required to compute the query is similarly limited.

On the other hand, if position samples are used to generate the index, then selecting the particular synchronization point used to access the compressed data given a query is straightforward, since the $\lfloor x/\rho \rfloor$ th pointer assigned to *rname* is the one required, without there being any need to search for it. But now the extent of the secondary sequential search to identify the first read aligned within the desired range is not limited by the parameter chosen, and may vary considerably, depending on the coverage of the data in question.

In the case of CSAM the aim is to create an index that permits fast extraction of reads and their related SAM fields. With this in mind, CSAM employs an index using a mixture of the two basic methodologies, storing a position sample index for querying the fields RNAME, POS, and SEQ, so as to achieve fast access to these components (see Section 3.1 of the supplementary material), assuming that the time used to search for a position interval using a position sample scheme would be less than the time used to find the desired synchronization points and then search for the required data, if block sample or the BAM approach to create the index were used. In addition, to minimize unnecessary data extraction during sequential search, CSAM creates a separate index for the QUAL field and the remaining SAM attributes. This allows the information contained in these fields to be used as extra conditions that augment *getInterval* queries; for example, once the desired data range is found, to select the reads within the range that have a mapping quality higher than some value specified by the user. The information stored in these fields does not need to be accessed until the read sequence data has been located. Finally, to connect the position sample index of the read sequences with the block sample index of the remaining fields, CSAM stores an extra value per synchronization point in the position sample indicating the number of data lines stored through until that point. That is, each position sample contains a tuple (n, d) where n is the number of lines up to the position sample, and d is a pointer to the location of the beginning of the read information for that position.

As presented CSAM uses two indexes (block sample and position sample index) over the compressed data in order to provide random access to the information stored. Query time performance thus depends on the sample parameters chosen (λ and ρ respectively). An example of how CSAM uses these two indexes to support *getInterval* operations is shown in Figure 3, and described by the following scenario. If $\rho = 1000$ for the position sample index, $\lambda = 500$ for the block sample index, and the query is "obtain all the data lines where RNAME is *chr*20 (chromosome 20), the reads start between position (POS field) 1500 and 2500, and its CIGAR string contains at least 10 soft clippings", then the extraction process consists of:

- Assuming that the reference and interval are valid, get entry $\lfloor 1500/\rho \rfloor$ from the position sample index for RNAME "*chr*20" which gives d, the byte address where reads start for the block containing POS 1500, and n, the number of lines that occur before d.
- Sequentially process lines from the compressed reads beginning at d until its POS value is higher or equal to 1500, keeping a count of lines read, c.
- Look up for the block sample index at position $\lfloor \frac{n+c}{\lambda} \rfloor$, which gives the disk location at which the data associated with the remaining fields is stored.
- Finally, extract data from all the fields until the value of POS in an extracted line is higher than 2500, only keeping lines in which the CIGAR string contains at least 10 soft clipping. This step is performed while that the information extracted fits in RAM. If it gets too large, the information is output, and the loop continued.

In the next section we present the results obtained when a range of test files are compressed and random-accessed using CSAM, comparing the trade-offs offered CSAM against the attributes and performance of existing SAM compression mechanisms.

4 Results

Table 2 in the supplementary material lists the 13 test files used in our experiments, varying in size from 1.4 GB to 21.7 GB. All experiments were performed on a computer with Intel(R) Core(TM) $i7$-2600 processor up to 3.40 GHz, 4 GB of main memory, 8 MB of cache, and HDD secondary storage. The operating system was Ubuntu 12.04, version 3.2.0-70-$generic$ Linux kernel. The CSAM methods described here were implemented in C++, using version 4.6.3 of the g++ compiler.

4.1 Lossless Compression

A key aim of CSAM is to attain a high level of lossless compression. To compare CSAM against BAM we record the space associated with the corresponding compressed formats for each of the test files, and also measure encoding and decoding throughput. Compression and decompression times were taken as the mean of ten consecutive runs for each file, after an initial run to prime the cache memory. We also computed the standard deviation of the 10 runs. The same methodology was used to measure compression and throughput for two general purpose compressors, gzip and bzip2. The CSAM approach requires that index parameters be chosen, and we used $\rho = 1000$ and $\lambda = 1000$ to determine the intervals between the synchronization points. Table 3 (in the supplementary material) lists the command lines and parameters used for the compression methodologies used in the experimental process. We also explored the lossless compression modalities of CRAM and NGC, both of which require the appropriate reference sequence to be supplied as part of the encoding and decoding protocols. The reference sequence is external to the file being compressed, and is not counted as part of the compressed size, a distinction which favors these two methods in the comparisons we carried out. Furthermore, as explained before, these two implementations do not offer lossless compression.

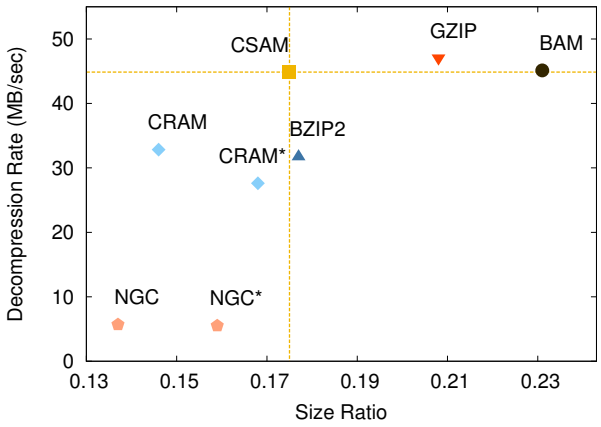

Fig. 4. Average decompression time versus average compression ratio using different lossless compression approaches over the 13 test files. CRAM* and NGC* represent the two schemes when the OTHER field is included compressed with gzip.

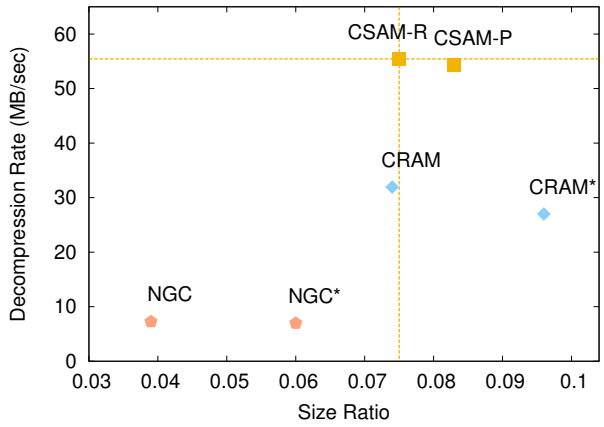

Fig. 5. Average decompression time versus average compression ratio using different lossy compression approaches over the 13 test files. CRAM* and NGC* represent the two schemes when the OTHER field is included compressed with gzip. included.

Figure 4 plots average compression ratio (that is, compressed size divided by original size of the file) on the horizontal axis and average decompression rate on the vertical axis, for all of the methods examined. In this figure, the "desirable" quadrant is the upper left region. If fully lossless (rather than information preserving) compression is required, then CRAM and NGC need to include all the information of the unstored fields, instead of recomputing it. To give an idea of the extra space involved, the cost of using gzip –9 to store the OTHER field (in general, the largest field among the ones recomputed by CRAM and NGC) is used to compute two further data points, shown as NGC* and CRAM*. Tables containing all test results are provided in the supplementary material.

Of the methods plotted in Figure 4, NGC gives the best compression. The drawback of NGC is its throughput, being 6 to 8 times slower than the other compressors. At the other end of the spectrum, gzip and the gzip-based BAM method provide high throughput, but with reduced compression effectiveness compared to NGC and CRAM. Between these two endpoints, bzip2, CSAM, and CRAM offer further options. Amongst the non-indexed lossless methods, CRAM offers a strong mix of attributes, with similar compression to NGC, and throughput rates close to those obtained using bzip2. The two drawbacks of CRAM are that it needs the reference sequence as extra input to the decompression and compression process, and that it does not fully losslessly compress all the SAM fields, recomputing them on decompression instead. A further distinction to be noted is that CSAM and BAM both include index information so as to support random access to the underlying data. CSAM uses notably less space than BAM, but BAM offers faster compression (see the supplementary material for details). Both methods offer similar decompression throughput. CSAM also treats each field separately, enabling random access to and decompression of individual fields, without needing to access whole blocks of data as in BAM. In addition – discussed in the next subsection – CSAM includes two lossy compression modes which greatly reduce the space required, at the cost of reduced fidelity of the quality scores.

4.2 Lossy Compression

The two lossy techniques for handling quality scores are referred to as CSAM-P and CSAM-R, using Cánovas *et al.* (2014)'s P-Block and R-Block quality compression techniques respectively. We set the parameters of the two schemes empirically, choosing the values the $p = 6$ and $r = 1.4$ for P-Block and R-Block respectively, which gives the best compression while maintaining at least 99.0 per cent recall and precision in the variation-call downstream application (Cánovas 2015).

CRAM and NGC provide lossy compression over the QUAL field too, and also the option to dispense completely with some of the fields. We do not explore the latter, and restrict our experiments to the CRAM and NGC compression parameters that provide lossy compression of the QUAL field, with all remaining fields losslessly compressed. All CRAM results reported in this section use a lower bound mapping quality of 50 and the Bin-Preserve mode, which were the settings that offered the best trade-off in preliminary experiments. Popitsch and von Haeseler (2013) provide a detailed experimental study of NGC in their supplementary material[9], and demonstrate that the modality *m1* was the only one that assured over 99.0 per cent of the original variants were recovered following lossy compression.

Figure 5 contrasts average throughput and average compression ratio for the lossy methods, using the same methodology as in Figure 4. NGC again offers the highest compression rates, but with the same drawbacks in terms of the compression and decompression rates, and in terms of requiring that the reference sequence be made available. Similarly, CRAM remains an attractive option if the primary functionality required is full compression and decompression of SAM files, and if the space used to store the reference sequence is not a determining factor. On the other hand, the CSAM-P and CSAM-R methods, while using slightly more space than CRAM, also support random access to the data, are independent of any external reference sequence, and allow fast compression and decompression compared with the other lossy compression techniques shown. Tables detailing these results across the individual test files are provided in the supplementary material.

4.3 Random Access

Like BAM, the CSAM compressor allows random access to the stored data. In particular, *getInterval* queries (Section 1.3) are handled without decoding the entire file. To resolve these, CSAM uses two indexes (block sample and position sample) over the compressed data; we use a parameter value of 1000 for both. Those parameters were chosen as an outcome of preliminary experiments, and provide a useful trade-off between throughput and index size.

[9] `http://nar.oxfordjournals.org/content/suppl/2012/10/11/gks939.DC1/nar-01707-met-n-2012-File002.pdf`

In order to test this operation the experiments considered two parameters for generating the queries: the number of occurrences extracted for a given interval (*nc*), and the number of queries to be executed (*nq*). For example, if $nc = 100$ and $nq = 1000$, then the experiment consisted of 1000 randomly selected intervals, with the corresponding reference name from the original SAM file, with 100 lines contained in the interval. Each query batch was then ordered by reference and start position, to facilitate "elevator"-style processing. For each of the methodologies tested, the mean time (including the standard deviation) to extract each line was measured by running sets of different sample intervals. Each experiment was run 10 times, after an initial run to prime the cache memory. The implementations used were: the lossless CSAM; the two lossy CSAM variants used in (CSAM-P with $p = 6$ and CSAM-R with $r = 1.4$); and the SamTools software, used to compute *getInterval* over pre-indexed BAM format files. All of these methods receive an input file consisting of 3 columns (reference name, start position, end position) that are the queries to be executed sequentially. Note that this methodology reflects a simple version of the *getInterval* query, and that it is possible to add more specifications, such as minimum mapping quality desired, number of soft-clipping bases, and so on. While SamTools allows input from a file containing all queries to be processed, it is also possible to run the queries in a sequential, brute force form, where each query is given as a command line parameter to the tool, and the program is restarted each time. A minor drawback of running the queries sequentially is that SamTools is not able to detect overlapping intervals, returning a SAM file which could contain duplicate information.

Figure 4 in the supplementary material plots average per-line retrieval times against the compression ratios measured for each of the 13 test files, for four combinations of *nc* and *nq*, with *nq* and *nc* set to the same values so as to obtain a cross-section of the overall performance profile. Note the differing vertical scales in each of the panes in the figure; substantial economies of scale arise when multiple lines are fetched from the same region of the compressed file. The CSAM approaches offer good overall performance in three of the four panes, but provide slightly slower per-line average access when $nq = nc = 10{,}000$. As in the previous plots, the lossless version of CSAM uses slightly less space than BAM, and the lossy CSAM variations (which still have relatively minor loss of information) use less than half the space of the BAM files.

Also evident in supplementary Figure 4 is that the greater the number of queries and the number of occurrences extracted per query, the narrower the band of measured execution times. This trends arises because the fraction of time spent searching is becoming smaller, while the fraction of time spent on sequential decoding of lines is becoming larger.

4.4 Downstream Application: Feature Count

Section 1.4 noted that computing coverage – the number of reads stored – over an interval is an important step in the analysis of genomic data, often focusing on user-defined desirable chromosome intervals. In this section we examine the use of the featureCounts mechanisms (Liao *et al.* 2014), which is widely used for this purpose.

When featureCounts evaluates a query, it searches over the aligned read information in which the condition indicated by the genomic features are fulfilled. All queries start by indicating an interval, and then optionally add more conditions. In this process not all the SAM fields need to be examined, with the set required depending on the genomic features being queried. We define *Simple* SAM *Notation* format (SSN) as a SAM file in which fields not involved in the featureCounts computation are replaced by empty values, rather than being extracted when the subset SAM file is computed. We mimic the same process by introducing the *getIntervalSSN* operation, which is similar to *getInterval* but with an extra parameter to indicate which SAM fields are to be extracted.

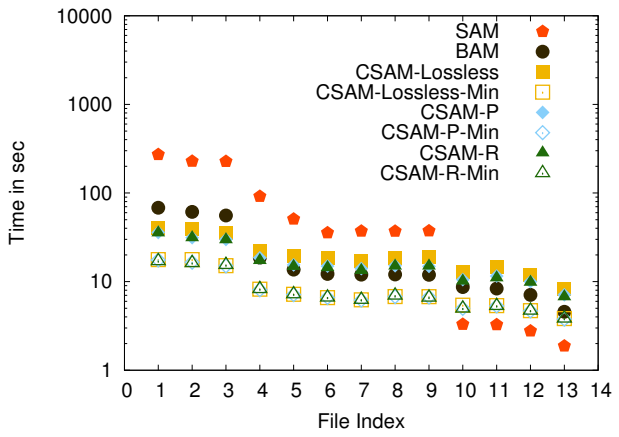

Fig. 6. Time to compute featureCounts over `hg19_RefSeq_exon.saf` and the files listed in Table 2 in the supplementary material, using SAM, BAM, and our CSAM approaches. The Min versions of CSAM indicate cases for which the SSN file generated contains only the main SAM fields (FLAG, RNAME, POS, MAPQ, CIGAR and SEQ fields) needed to calculate coverage.

Using this function and the list of genomic features, we generate a subset SSN file containing only the aligned reads whose start positions are relevant to the desired query intervals. Finally this file, in addition to the list of genomic features, is given as input to featureCounts, generating the same output as when the original uncompressed SAM file is used.

In order to assess the performance of CSAM in regard to featureCounts, we measure execution time including the generation of the SSN file, using CSAM's *getIntervalSSN* operation, and including the cost of running featureCounts on the SSN data that is generated. For SAM and BAM, the corresponding cost is measured as the time taken by featureCounts when supplied with a full data file, plus the query. We use the same data sources as Liao *et al.* (2014), namely the exon locations for the human reference *hg19*, and compare the SAM, BAM, and CSAM (the lossless version and two representative lossy versions, CSAM-P with $p = 6$ and CSAM-R with $r = 1.4$) storage format. More information about the structure of these experiments is provided in the supplementary material.

Figure 6 compares featureCounts times across the range of file formats and across the suite of test files. Note that the vertical axis is logarithmic and the files are ordered in the horizontal axis by decreasing SAM file size order, which leads to large query-time differences between files 3 and 4 (a 12 GB difference in size between them); between files 4 and 5 (a 2 GB difference in size); between files 9 and 10 (a difference of 1 GB); and between files 12 and 13 (another 1 GB).

The description given by Liao *et al.* of how featureCounts operates with SAM files suggests that it searches over all the aligned reads to identify the ones with the specified genomic features. The timings shown in Figure 6 support that observation – the larger the input SAM file is, the longer the time taken to finish the process of counting the coverage for each genomic feature. For small size SAM files (files 10 to 13 in Table 1), featureCounts offers a faster alternative, giving that the time used decompressing other format files information is longer than completely reading the SAM file. Meanwhile, when the BAM or the CSAM approaches are used as input, the time required by featureCounts oscillates, with BAM generally giving slightly better time performance for low coverage and CSAM slightly faster for high coverage files.

We also measured three CSAM-Min variations, which decode only the required fields. They provide the fastest approach for most of the test files, the exception being when the SAM files are small, in which case using the original SAM file as input for featureCounts is faster. We also

experimented with further variations that generated different combinations of fields. The only case in which the times obtained were more than slightly altered was when the QUAL field or the OTHER field were also added to the decoded output. Neither of these fields are employed in any of the operations currently provided as part of featureCounts.

Beside comparing the time obtained using featureCounts with SAM, BAM, and CSAM, it is also important to consider the storage space used by each of the input files. From the results presented in the previous sections it can be observed that BAM uses on average around 23.1%, CSAM-Lossless 17.5%, CSAM-P (with $p = 6$) 8.3%, and CSAM-R (with $r = 1.4$) 7.5% of the space used by the original SAM file. Note also that, compared with SAM and BAM, in these experiments the CSAM approaches were used as pre-processor for a stand-alone featureCounts process. That is, the generated SSN file was written to disk and then passed as input to featureCounts, at which point it was opened, re-read, and analyzed. This double-handling could be avoided if featureCounts was modified so that the CSAM format was handled directly.

5 Conclusion

We have described CSAM, a compressed and indexed representation for genomic data files originally produced in SAM format, which supports queries over the stored information without requiring whole files to be decompressed. Each SAM field is treated individually, with a focus on compressing the Read Sequences and QUAL fields, as those two fields use the largest amount of space amongst the compulsory SAM fields. CSAM does not make use of any external reference file, making it self-contained, and allowing decompression to take place without access to a library of reference sequences. In both the CRAM and NGC methods it is necessary to store the particular reference sequence that was used to compress a file to guarantee that it can be decompressed accurately.

Our experimental results show that CSAM provides a useful balance of attributes, and can be applied in situations in which other compressed formats may be inappropriate. Much of the gains that have been achieved are a consequence of treating the 11 SAM fields independently, and then exploring options tailored to the particular symbol alphabets and distributions encountered. These analyses to date have been based solely on the statistical textual properties of the components, without considering the biological meaning of the data. For example, we did not study how the compression of the SEQ field could be improved if indels, soft clipping, paired ends reads and so on were employed, nor make use of any other information available from the SAM fields. Including these factors could lead to higher compression effectiveness at the cost of requiring more compression-time analysis and, potentially, lower decoding throughput. Also, given that in CSAM each field is treated separately, it should be possible to compress and decompress some of these fields in parallel which might lead to improved performance. What is clear is that the format of genomic files will continue to evolve, and hence applying separate compression methods to each field should continue to be a useful strategy.

An interesting future research topic would be to explore other uses of stored genomic data, and support these functionalities within the compressed data format. For example, it would be desirable to be able to find variations and output the respective VCF files without the need for any external tool. Since the core of this work was completed (Cánovas 2015) new ideas have continued to emerge (Bonfield 2014; Grabowski *et al.* 2015; Hach *et al.* 2014; Alberti *et al.* 2016; Hernaez *et al.* 2016; Roguski and Ribeca 2016); comparing the performance of these new methods against CSAM will almost certainly lead to further insights as to how genetic data can best be stored.

Acknowledgment

This work was supported by the NICTA Victorian Research Laboratory, and funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Center of Excellence program. We thank Vadim Zalunin for helping with the CramTools usage; and Wei Shi and Jan Schröder for sharing their knowledge of the area.

References

Alberti, C., Daniels, N., Voges, J., Goldfeder, R. L., Hernandez-Lopez, A. A., Mattavelli, M., and Berger, B. (2016). An evaluation framework for lossy compression of genome sequencing quality values. In *Data Compression Conference (DCC)*. To appear.

Anders, S., Pyl, P. T., and Huber, W. (2015). HTSeq - A python framework to work with high-throughput sequencing data. *Bioinformatics*, **31**(2), 166–169.

Ansorge, W. (2009). Next-generation DNA sequencing techniques. *New Biotechnology*, **25**(4), 195–203.

Bonfield, J. K. (2014). The Scramble conversion tool. *Bioinformatics*, **30**(19), 2818–2819.

Cánovas, R. (2015). *Practical Compression for Multi-Alignment Genomic Files*. Ph.D. thesis, The University of Melbourne, Australia.

Cánovas, R. and Moffat, A. (2013). Practical compression for multi-alignment genomic files. In *Proc. 36th Australasian Computer Science Conference*, pages 51–60.

Cánovas, R., Moffat, A., and Turpin, A. (2014). Lossy compression of quality scores in genomic data. *Bioinformatics*, **30**(15), 2130–2136.

Church, G. M. (2006). Genomes for all. *Scientific American*, **294**(1), 46–54.

Cock, P. A., Fields, C. J., Goto, N., Heuer, M. L., and Rice, P. M. (2010). The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, **38**(6), 1767–1771.

Ewing, B. and Green, P. (1998). Base-calling of automated sequencer traces using Phred. II.Error probabilities. *Genome Research*, **8**(3), 186–194.

Ewing, B., Hillier, L., Wendl, M. C., and Green, P. (1998). Base-calling of automated sequencer traces using Phred. I. Accuracy assessment. *Genome Research*, **8**, 175–185.

Fritz, M. H., Leinonen, R., Cochrane, G., and Birney, E. (2011). Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, **21**(5), 734–740.

Grabowski, S., Deorowicz, S., , and Roguski, Ł. (2015). Disk-based compression of data from genome sequencing. *Bioinformatics*, **31**(9), 1389–1395.

Hach, F., Numanagic, I., and Sahinalp, S. C. (2014). DeeZ: reference-based compression by local assembly. *Nature Methods*, **11**, 1082–1084.

Hernaez, M., Ochoa, I., and Weissman, T. (2016). A cluster-based approach to compression of quality scores. In *Data Compression Conference (DCC)*. To appear.

Lawrence, M., Huber, W., Pages, H., Aboyoun, P., Carlson, M., Gentleman, R., Morgan, M. T., and Carey, V. J. (2013). Software for computing and annotating genomic ranges. *PLoS ONE*, **9**(8), e1003118.

Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., and Durbin, R. (2009). The sequence alignment/map format and SAMtools. *Bioinformatics*, **25**(16), 2078–2079.

Liao, Y., Smyth, G. K., and Shi, W. (2014). featureCounts: an efficient general purpose program for assigning sequence reads to genomic features. *Bioinformatics*, **30**(7), 923–930.

Mardis, E. R. (2008). Next-generation DNA sequencing methods. *Annual Review of Genomics and Human Genetics*, **9**(1), 387–402.

Myllykangas, S., Buenrostro, J., and Ji, H. P. (2012). Overview of sequencing technology platforms. In *Bioinformatics for High Throughput Sequencing*, pages 11–25. Springer New York.

Popitsch, N. and von Haeseler, A. (2013). NGC: lossless and lossy compression of aligned high-throughput sequencing data. *Nucleic Acids Research*, **41**(1), e27.

Richterich, P. (1998). Estimation of errors in "raw" DNA sequences: A validation study. *Genome Research*, **8**(3), 251–259.

Roguski, Ł. and Ribeca, P. (2016). Cargo: effective format-free compressed storage of genomic information. *Nucleic Acids Research*.

Wan, R., Anh, V. N., and Asai, K. (2012). Transformations for the compression of FASTQ quality scores of next-generation sequencing data. *Bioinformatics*, **28**(5), 628–635.