Supplementary Methods for the manuscript "FaStore – a space-saving solution for raw sequencing data"

Łukasz Roguski, Idoia Ochoa, Mikel Hernaez, Sebastian Deorowicz

Contents

1 Methods description

1.1 Compression of DNA Sequences

The general data processing workflow and DNA sequence compression algorithm is based on ORCOM [**?**] with some major improvements. First, FaStore introduces a "paired-end" compression mode, which allows to preserve the pairing between the reads when sequenced in the paired-end configuration. In this mode, the pairing between the FASTQ reads is preserved with the sequences and the decompressed reads will reside on the same lines in the output files (while sharing the same read identifier, possibly differing only by the indicator marking the number of the read in the pair). Second, FaStore introduces a number of optional reads re-distribution steps to improve the DNA sequence compression ratio. Hence, the standard DNA sequence compression mode corresponds to the one as in ORCOM and is denoted as "C0" mode. The extended one, consisting of multiple reads re-distributon stages is denoted as "C1" and is the default one. Nonetheless, the main underlying ideas behind processing of the reads and compressing the DNA sequence remain similar as in ORCOM.

To exploit the significant sequence redundancy present in high-coverage HTS data FaStore aims to cluster the reads into groups, sharing a significant sequence overlap. This allows to achieve a high level of compression of the DNA sequences. The data processing workflow is divided into two (or optionally three) main phases. First, the reads from FASTQ file(s) are distributed into independent bins in which they are clustered. Optionally, in the next step, the reads follow an additional re-distribution phase in order to improve their clustering. Finally, reads inside bins are compressed. The decompression algorithm consists of the reverse of the compression phase. Since the bins can be processed independently, the compression and decompression algorithms are highly parallelizable. A more detailed description of each of the phases is presented in the following subsections.

1.1.1 Signatures selection

Let $s = s[0]s[1]...s[n-1]$ be a string of length n over a finite alphabet Σ of size σ. Given $0 \leq i \leq j < n$ the symbol $s[i]$ denotes the $(i+1)$th symbol of s, the $s[i \dots j]$ denotes the substring $s[i]s[i+1] \dots s[j]$, and the $s \circ t$ denotes the concatenation of strings s and t.

The *minimizer* for a read with a sequence s of length r is the lexicographically smallest of its all $(r-p+1)$ p-mers, where usually $p \ll r$. In a very simplified case, such minimizer can be already used as an identifier of the bin into which the read will be placed. The motivation here is that if two sequences share a large overlap, they are also likely to share the same minimizer.

Moreover, for practical reasons, we assume the alphabet size $\sigma = 4$, i.e., $\Sigma = \{A, C, G, T\}$, which will give a total of σ^p possible minimizers. Through a significant number of preliminary experiments we found a reasonable length of p-mer to be $p = 8$ (the default parameter), which gives $\sigma^8 = 65536$ possible minimizers. This way, each minimizer can be directly mapped to an integer value, encoding each symbol using 2 bits. However, since we only consider 4 possible DNA symbols, all the minimizers containing at least one unsupported symbol N are mapped

to a special bin, labeled N. Hence, the maximum number of independent bins is $4^8 + 1$.

When searching for minimizers, as DNA sequences can be read in two directions (forward and reverse with complements of each nucleotide), we process each sequence twice in its original and reverse-complemented form. Additionally, we introduce a "skip zone" z, that is, not to look for minimizers in the read suffixes of a given length (default: $z = 12$ symbols in "C0" mode or $z = 0$ in "C1" mode). Therefore, the minimizers are sought over $2 \times (r - z - p + 1)$ resulting p-mers, where r is the length of the read. We call them *canonical minimizers*. The introduction of the skip zone will be useful when performing reads matching (as will be explained in the following subsections).

However, a problem with strictly defined minimizers is an uneven distribution of bins, where the bins identified by minimizers starting with or including repetitions of identical nucleotides will also contain the majority of all the reads. In ORCOM we restricted the canonical minimizers by excluding the triplets AAA, CCC, GGG, TTT (apart from containing any 'N' symbol). However, we found that the restrictions on canonical minimizers posed by KMC2 [**?**] tend to provide a better reads distribution in the bins, hence, we also decided to use them in FaStore. These restrictions are: not to contain AAA subsequence, neither to start with ACA and neither to contain AA anywhere except at the beginning of the minimizer. Such restricted canonical minimizers we denote as *signatures*.

1.1.2 Distribution of the reads into buckets

When processing reads from FASTQ file(s) we first search for their signatures, which will be used as an identifier of the bin. As previously mentioned, we search for the signature in the DNA sequence in two directions – forward and backward with a reverse-complement of each base. From the two generated signatures, we select the lexicographically smallest one and keep the information about the direction in which it was found.

There are, however, some differences in handling the reads between the single- and paired-end mode. In single-end mode, we search for the signature only in the read's sequence itself and place the read in the corresponding bin. In the paired-end mode, we search for signatures in both sequences of a pair. Having determined the two signatures with their direction indicators, we select the lexicographically smallest one (and its direction) and keep the information which read of the pair it originates from – this read we will call as the *main* read. When processing reads in the single-end mode the situation is straightforward – all the reads are treated as main reads. Finally, the read or the pair of reads is placed in a bin identified by the determined signature. If no valid signature could be found in the read(s), it (or the pair) is placed in the special 'N' bin.

During the read distribution stage, the reads are analyzed and some statistics related with base quality scores and read identifiers (if these are used) are gathered. The details are given in the following subsections. Having the reads distributed into bins, we proceed to reads-matching followed by an optional reads re-distribution stage.

1.1.3 Building reads matching trees

Independently for each bin, we reorder the reads inside the bins to move overlapping reads possibly close to each other. The operation is similar to the reads encoding stage performed in ORCOM, yet, with significant improvements. First, we sort the reads (sequences) s_i, for all i, according to the lexicographical order of the string $s_i[j \dots r-1] \circ s_i[0 \dots j-1]$, where j is the position of the signature for the s_i. Here is when setting the appropriate skip zone length z becomes relevant, as it will influence the matching of the reads. When z is set too small, some signatures will be found close to the end of the sequence. The first factor of the sorting criterion, i.e., $s_i[j \dots r-1]$ will be too short to have a good chance of placing the read among those that overlap it in the genome and, hence, finding the read matches. On the other hand, with the zone being too long many possibly overlapping sequences will be forbidden.

With the reads sorted, we proceed to match them. During this step, we will be building a graph G over the reads, composed of independent sub-graphs – trees, where each read with its sequence s_i is indicated by some node v_i. When matching, we maintain a buffer (a window) of m previously processed reads (by default $m = 256$) and for each read s_i, we seek a read from the buffer which maximizes the overlap with s_i.

The distance between a read and a considered match depends on the number of elementary operations transforming one into another. For example, given two reads with their sequences respectively *AACGTacgaCGGCAT* and *CCTacgaCGGCATCC*, where *acga* denotes a signature, we match them after a (conceptual) alignment:

```
AACGTacgaCGGCAT
  CCTacgaCGGCATCC
```

to find that they differ with one mismatch ('G' versus 'C') and 2 end symbols of the second sequence have to be inserted, hence the distance is $d = c_{\mathrm{m}} \times 1 + c_{\mathrm{i}} \times 2$, where c_{m} and c_{i} are the mismatch and the insert cost, respectively. The default values for the parameters are $c_{\mathrm{m}} = 2$ and $c_{\mathrm{i}} = 1$, and they were chosen experimentally. The read among the m previous ones that minimizes such a distance, and is not greater than the maximum distance threshold $d_{\max}$ (set by default to a half of the read length), is considered a reference for the current read. There are three possible results of such defined matching.

A read for which no referential read was found is classified as *hard* read. A read for which a referential read was found with the distance $d = 0$ is classified as *exact match*. Otherwise, the read is classified as *normal match*.

Moreover, if no referential read was found for the read s and the advanced search mode is used ("-r" flag in the command-line parameter of FaStore, set as default in "C1" mode), we perform an additional search for the potential matches of s in the reads processed so far. Let the $s[j \dots i]$ be the reverse of the sequence $s[i \dots j]$, where $j > i$. To facilitate the search we keep an auxiliary buffer of all the reads processed so far sorted by their inversed prefix $s_i[j-1 \dots 0]$, where j is the position of the read's s_i signature. We compare lexicographically each read s_i with s, starting from their positions of minimizers, i.e., we compare $s[j-1 \dots 0]$

with $s_i[j'-1\ldots 0]$. This additional search step allows to reduce the number of reads classified as hard reads, which could have been reported as such due to a sequencing error present in the read just after the signature. In such cases, the reads after sorting could be distant by more than m positions and would not be matched by using a reads buffer keeping only m previous reads.

Finally, having the read s (with its corresponding and unlinked node v) classified and having its possible referential read s_k (with node v_k), we update the graph G and the auxiliary read matching buffers. If the read was classified as a hard read then we create a new tree T with its root node T_v set to v. Otherwise, the read's node is set as a child node of the referential node v_k, keeping the information about matching. Such constructed trees can be though as small clusters of reads sharing a high degree of sequence similarity between them.

Important to mention, in the paired-end mode, the above matching is performed only for the main read. The matching of the second read from the pair is performed in a different way (see Subsection 1.1.7).

1.1.4 Re-distribution of the reads

With the reads initially distributed into bins, we perform a further re-distribution of the reads in an iterative way. The goal is to create larger clusters of highly similar reads to possibly bring the reads from the same genomic regions close to each other. This operation should also improve the results of paired reads matching.

Let Y be the initial set of valid signatures represented as integer values. Let f be a signature filtering parity parameter (by default, at the beginning $f = 1$). We define a new set of valid signatures $\hat{Y}$, being a subset of the initial Y, such that $\hat{Y} = \{\hat{y} \in Y : \hat{y} \mod 2^f = 0\}$. For each bin identified by the signature $y \in Y \setminus \hat{Y}$, we process all of its stored trees as follows.

For each tree T we traverse it starting from its root T_v and look for a pair of reads residing at the "edges" of the possible genomic region that the tree T covers with its reads, i.e., the reads with the extreme positions of the signature. These reads must also contain a new signature from $\hat{Y}$. We look for one read that has its new signature possibly close to the beginning of its sequence (j close to 0) and for another one that has its new signature close to its end (j close to $r-1-p-z$). From these reads, we select the one which has the smallest signature $\hat{y}$. The node v corresponding to this read will be a new root node of T. Next, with a new T_v we update T and move it into a bin identified by $\hat{y}$. However, if no new valid signature was found for any of the reads in T, we leave the tree in the current bin intact. Once all the trees have been processed, we perform reads sorting and matching as explained in the previous step.

What's important to note, although for each moved tree T, its root's read sequence s (as pointed by the T_v) contains the signature $\hat{y}$ of the new bin, the other reads present in T may not. Therefore, when moving a tree T to a new bin, we practically only move its root node, which keeps the information about the tree structure. Such read (node) will be processed as a regular read during reads matching stage and the information related with the kept sub-tree

will be used during the final compression stage. As a consequence, when performing multiple redistribution steps, one node can store information about multiple subtrees.

Finally, in the next step, we either proceed to build contigs based on the built trees or follow another step of reads re-distribution, increasing the parameter $f \leftarrow f + 1$ and setting the initial set of signatures Y as $\hat{Y}$. As a side note, usually, performing more than 3 reads re-distribution steps does not give any significant improvements in compression ratio, hence, in "C1" mode we perform only 3 steps.

1.1.5 Assembling contigs

Having built the final reads matching graph G we proceed to assembly contigs, independently per each bin and independently per each tree inside the bin. A contig C is defined by a set of reads (nodes) assembling a consensus sequence C_{seq}, its length C_{len} and a set of possible variants C_{var} in the consensus. It is later linked with G through its main node C_v. Given the maximum length of reads $r_{\max}$, the maximum length of the consensus sequence is $2 \times r_{\max}$[1], where the "center" of the consensus resides at the position $r_{\max}$. The "center" of the consensus corresponds to the sequence $C_{\text{seq}}[r_{\max} \dots r_{\max}+p-1]$, which will be used to anchor the reads to C_{seq} using their signatures. Therefore, the sequence $C_{\text{seq}}[r_{\max} \dots r_{\max}+p-1]$ will be constant, shared across all the reads building the contig.

For each tree T, we traverse T from its root node following the breadth-first-search method. We start with an empty contig C and process the T nodes as follows. Let v be the next node we visit. If the current contig C is empty, then we add the read s (as indicated by v) to C and select v as the main node of C denoted as C_v (it will be later used for linking C with T). Otherwise, we try to add s to the contig by firstly checking whether the read will introduce new variants into the consensus sequence, as these will need to be also shared (and encoded) by the rest of the reads in the current contig.

When performing this check, for the read s we analyze how $s[c \dots j-1] \circ s[j+p \dots r-t-1]$ matches with the C_{seq} (anchored using its signature position), where c is the sequence trimming length (by default: $t = 2$) and r is the read length. In a number of experiments, we found that by skipping the first/last t bases from the read during the sequences comparison, we managed to build better contigs, i.e., consisting of a higher number of reads and with a smaller number of introduced variants.

To show an example, given a read *AGAacgaCGGCATCC* and a contig with a consensus sequence *TCAAACGTacgaCGGCAT* constructed from two reads (*CCGTacgaCGGCATT* and *TTCAAACGAacgaCG*), where *acga* denotes a signature, we perform a (conceptual) alignment, similarly, as when matching the reads:

[1] Actually, $2 \times r_{\max} - p$, where p is the length of the signature and also the minimum length of the possible overlap between two reads with the signatures designated on their opposite ends. However, for simplicity, we assume $2 \times r_{\max}$.

```
        CCGTacgaCGGCATT
    TTCAAACGAacgaCG
---TCAAACG*acgaCGGCAT-----
          AGAacgaCGGCATCC
```

The new read aligns with the consensus sequence with two mismatches. The first mismatch is at the beginning of the read ('A' vs. 'C'), but it is discarded (with a sample $t = 1$). The second mismatch ('A' vs 'T' / "*") corresponds to the variant in the consensus sequence, so the read does not introduce any new ones. In this simple example, the read will be added to the contig and will update the consensus sequence – it will add the 'C' present at its penultimate position (discarding the last 'C' due to trimming parameter). The updated consensus sequence will be *TCAAACGTacgaCGGCATC*.

However, in a more realistic scenario, before accepting the read s in the contig C we analyze few other conditions. With the read anchored to the consensus sequence C_{seq}, we calculate the Hamming distance d_h between the two sequences (taking also into account the length of sequence trimming t). That is, we calculate $d_h = \text{Hamm}(s[t \dots j-1] \circ s[j+p \dots r-t], C_{\text{seq}}[r_{\max}-j+t \dots r_{\max}] \circ C_{\text{seq}}[r_{\max}+p \dots r_{\max}+r-j-t])$. During this operation we also check for a number of potential new variants d_{var} which the read may introduce to C_{seq}. If the calculated d_h is above the given threshold $d_{h\_\max}$ (by default: $d_{h\_\max} = 8$) or the number of new introduced variants is above $d_{\text{var\_max}}$ (by default: $d_{\text{var\_max}} = 1$), then the read is discarded. Introducing new variants by the read to the contig imposes that all the bases at the positions corresponding to the new variants (which are covered by the current reads in C plus the new one) will need to be encoded in a different and less efficient way (explained in the following section). Hence, in overall, it may be more convenient to encode the read separately as a regular referential match.

When no more reads can be added to the current contig C, we perform its post-processing. First, we remove the reads containing variants that are not present in any other reads. The aim is to reduce the cost of encoding the contig as a whole, since with a variant present only in one read, all the other remaining reads would need to handle it during the encoding stage. If the size of the filtered contig (the number of reads assembling it) satisfies $C_{\text{size}} \geq C_{\text{size\_min}}$ we accept the contig C as a valid one (by default: $C_{\text{size\_min}} = 10$). Otherwise, we discard it. With a valid C we refine its consensus sequence by selecting the most frequent bases appearing in the reads at respective positions (the majority voting).

Finally, all the nodes whose reads are assembling the contig C (apart from the main node C_v) are unlinked from the graph, as they will be encoded differently. The main contig node C_v stores the information about its structure (and reads) on a similar way as a node is keeping information about storing possible sub-trees. Note that multiple contigs can be built per one tree.

1.1.6 Encoding the reads

Having built the reads matching graph and the contigs we proceed to encode the reads. Independently per each bin, we traverse each tree T starting from its root node following the breadth-first-search method. Let v be the next node we visit and s the sequence of the read indicated by v. Firstly, we encode s according to the type of the match it was classified. If v is the root node of T we already know that the read was classified as a hard read (and it has no referential read). Otherwise, the read was classified either as a normal match or as an exact match using the read $\tilde{s}$ as a reference (indicated by the parent node $\tilde{v}$ of v). The differential encoding of s using $\tilde{s}$ as a reference is performed on a similar basis as in ORCOM with some modifications.

After encoding the sequence s, we check whether v stores any sub-trees originating from other bins (and reads of which use a different signature than the reads in the currently traversed tree T). In such cases, for each sub-tree $\tilde{T}$ of v we emit some control information about it and proceed to traverse it using v as a "virtual" root node. The reads of $\tilde{T}$ are referentially encoded as normal following the above described algorithm. After finishing encoding of the sub-tree $\tilde{T}$ we emit information about its end. The nodes in $\tilde{T}$ can also contain information about other sub-tress – the encoding of the sub-trees is a recursive operation where its maximum depth corresponds to the number of previously performed reads re-distribution steps.

Finally, we check whether v stores a contig. In such case, for contig C we emit some control information about it (such as the length of the consensus sequence and the positions of variants), referentially encoding the consensus sequence C_{seq} with respect to s. In the next step, we proceed to encode the reads which assemble the contig. We sort the reads in the contig by their signature position, which slightly helps with compression. Then we encode each of the reads. At the end, we emit information about finishing the encoding of C. The whole encoding process is summarized in Algorithm 1.

All the encoding information is emitted into a number of streams. The streams and the used encoding schemes are as follows.

ReadFlags. Stores the flag corresponding to the method of the encoding of the current read. The flags can denote the type of classified read during referential matching, i.e., exact matches ($f_{\text{read\_copy}}$), normal matches ($f_{\text{read\_shift}}$, $f_{\text{read\_full}}$, $f_{\text{read\_full\_exp}}$), and hard matches ($f_{\text{read\_hard}}$). Moreover, they also can denote that the read assembles a contig ($f_{\text{read\_contig}}$, $f_{\text{read\_copy}}$). More specifically, they mean:

- $f_{\text{read\_copy}}$ – the current read is identical to the previously encoded one,
- $f_{\text{read\_shift}}$ – the current read overlaps with some previously encoded read without mismatches (only trailing symbols are to be encoded),
- $f_{\text{read\_full}}$ – the current read overlaps with some previously encoded read with ≤ 4 mismatches,
- $f_{\text{read\_full\_exp}}$ – as in the case of $f_{\text{read\_full}}$, but with > 4 mismatches,

Algorithm 1 Encoding of a reads matching tree – the encode_tree() function

Input: Reads matching tree T
Output: Encoding information stored in a set of streams $\{O_i\}$
while $v \leftarrow$ next_node_bfs(T) **do**
  $s \leftarrow$ dna_sequence_of(v)
  if v is the root node of T **then**
    Encode s as a hard read
  else
    $\tilde{v} \leftarrow$ parent_node_of(v)
    $\tilde{s} \leftarrow$ dna_sequence_of($\tilde{v}$)
    Encode $\tilde{s}$ according to its classified matching type using $\tilde{s}$ as a reference sequence
  end if
  if v encodes any sub-tree **then**
    for all sub-tree $\tilde{T}$ of v **do**
      Store the offset of $\tilde{T}$ with respect to s
      Set v as a "virtual" root node of $\tilde{T}$
      encode_tree($\tilde{T}$)
    end for
  end if
  if v encodes a contig **then**
    $C \leftarrow$ contig_of(v)
    Store the C_{var} and encode C_{seq} with respect to s
    for all Read $\tilde{s}$ in C **do**
      Encode $\tilde{s}$ according to C_{var} using C_{seq} as a reference sequence
    end for
  end if
end while

- $f_{\text{read\_hard}}$ – the current read has no referential read,
- $f_{\text{read\_contig}}$ – the current read is a read in the current contig but it differs from the previously encoded one (which is encoded using $f_{\text{read\_copy}}$ flag).

ControlFlags. Stores the control flags used to initialize or finalize encoding of a group of reads. These are:

- $f_{\text{group\_tree\_start}}$ – the current read starts encoding a sub-tree,
- $f_{\text{group\_contig\_start}}$ – the current read starts encoding a contig,
- $f_{\text{group\_end}}$ – the current read ends encoding of the current group (sub-tree or contig).

Rev. (Used with all flags from 'ReadFlag'). Stores binary flags telling whether reads are in the reverse-complemented form as in ORCOM.

HardReads. (Used only if 'ReadFlag' is $f_{\text{read\_hard}}$). Here the hard reads are stored almost in verbatim form as in ORCOM – we omit only storing the signature, which is replaced by an extra symbol ('.').

Prev. (Used only if 'ReadFlag' is $f_{\text{read\_shift}}$, $f_{\text{read\_full}}$, or $f_{\text{read\_full\_exp}}$). Stores the distance offset *dist* to the referenced read, i.e., a number of reads encoded between the reference read and the current one. In case of exact matches, the read is stored as the next one after the referential read, hence, the information can be encoded using only one flag $f_{\text{read\_copy}}$. Since the most frequent *dist* value is 0, occurring in short runs, we use a form of RLE-0 (run-length encoding) to store the values. More specifically, we use a special symbol 'S0' (emitted as '0' byte value) to encode a singular occurrence of 0 value, and 'S1' (emitted as '1' byte value) to encode two consecutive occurrences of 0 values. Any other offset of value > 1 is emitted as $dist + 1$. Moreover, in contrast to ORCOM, we allow to store both 8-bit and 16-bit offsets, not having the strict limitation of the size of the matching window. To handle this case, we use another code. More specifically, if $dist < 255 - 1$, then we store it as a byte. Otherwise, we store a special symbol 'SN' (emitted as '255' byte value) and store the *dist* using 2 bytes. Nonetheless, usually the majority of offsets will have a value less than 255.

Shift. (Used only if 'ReadFlag' is $f_{\text{read\_shift}}$, $f_{\text{read\_full}}$, or $f_{\text{read\_full\_exp}}$). Stores the positional offset of the current read against its referenced read as in ORCOM. The offset can be negative.

MatchRLE. (Used only if 'ReadFlag' is $f_{\text{read\_full}}$). Stores information on mismatch positions using a form of RLE as in ORCOM. Namely, each run of matching positions (of length at least 1) is encoded with its length represented on 1 byte, and if the byte value is less than 255 and there are symbols left yet, we know that there must be a mismatch at the next position, so it is skipped over. "Unpredicted" mismatches are encoded with 0.

MatchBinary. (Used only if 'ReadFlag' is $f_{\text{read\_full\_exp}}$). Stores the mismatch positions as a binary mask. When the number of mismatches in the read is greater than n_{mism}, the produced

RLE runs of matching positions will be relatively short. Therefore, in such cases, storing the information about mismatches in an alternative binary form can improve the compression.

LettersX. (These are used only if 'ReadFlag' is $f_{\text{read\_full}}$ or $f_{\text{read\_full\_exp}}$). Stores all the mismatching bases $[b_0, \ldots, b_n]$ from the current read where at their corresponding positions in the referenced read are bases $[\tilde{b}_0, \ldots, \tilde{b}_n]$. In contrast to ORCOM, we use only one stream (versus 5 separate ones) to encode the mismatching bases b_i, where the corresponding $\tilde{b}_i$ base is used as a context. All trailing symbols from the current read beyond the referential match are encoded using 'N' symbol as a context.

TreeShift. (Used only if 'ControlFlag' is $f_{\text{group\_tree\_start}}$). Stores the offset of the sub-tree against the referential read. More specifically, it stores the offset between the current read's (as indicated by the "virtual" root node) signature position and the position of the signature, which is shared by the reads in the sub-tree. The offset can be negative.

ContigCoord. (Used only if 'ControlFlag' is $f_{\text{group\_contig\_start}}$). Stores the length of the consensus sequence encoded as a pair of relative positional offsets from its beginning and its end with respect to its center.

ContigVarRLE. (Used only if 'ControlFlag' is $f_{\text{group\_contig\_start}}$). Stores information on variants positions in the consensus sequence using the same encoding technique as in 'MatchRLE' stream.

ContigConsensus. (Used only if 'ControlFlag' is $f_{\text{group\_contig\_start}}$). Stores the consensus sequence differentially with respect to the referential read (indicated by the contig's main node), similarly, as when encoding mismatching bases in 'LettersX' stream. This is, it stores all the variable bases $[b_0, \ldots, b_n]$ from the consensus sequence where at their corresponding positions in the referenced read are bases $[\tilde{b}_0, \ldots, \tilde{b}_n]$. When encoding b_i base its corresponding $\tilde{b}_i$ base is used a context. As the consensus sequence will be usually longer than the single referential match, all the bases not covered by the match are encoded using 'N' symbol as a context.

ContigShift. (Used only if 'ReadFlag' is $f_{\text{read\_contig}}$). Stores the positional offset of the current read against the previous one encoded as a part of the current contig.

ContigLetters. (Used only if 'ReadFlag' is $f_{\text{read\_contig}}$). Stores all the possibly variable bases $[b_0, \ldots, b_n]$ from the current read where at their corresponding positions in the consensus sequence are bases $[\tilde{b}_0, \ldots, \tilde{b}_n]$. We encode the variable base b_i using the corresponding $\tilde{b}_i$ base as a context. The encoding is done similarly as in 'LettersX' and 'ContigConsensus' streams.

The streams are compressed using either a strong general-purpose compressor PPMd[2] or our variant of a range coder (RC). Namely, the streams 'ReadFlags', 'ControlFlags', 'Rev', 'MatchBinary' are compressed using RC of order-4, where context is formed of the four previ-

[2] `http://compression.ru/ds/ppmdj1.rar`

ous symbols. The streams 'LetterX', 'ContigConsensus', 'ContigLetters' are compressed using RC of order-5, where the context is formed of the four previous bases and a mismatching (or variant) base from the referential sequence. All the other streams are compressed using PPMd, using order-4 context model and memory up to 16 MB. As a side note, in practice, some of the streams are compressed interlaced, namely: 'ReadFlags' with 'ControlFlags', 'TreeShift' with 'ContigCoord' and 'ContigConsensus' with 'ContigLetters', which allows to improve a bit the overall compression. Also note that, in parallel, when encoding the DNA sequences, we also encode their corresponding read identifier and base quality scores emitting the information into other streams – the description of the compression methods can be found in Sections 1.2 and 1.3 respectively.

1.1.7 Matching and encoding of the paired reads

The previously performed steps of building matching trees, re-distributing reads, and building contigs were taking into consideration only the main reads. When compressing the data in the single-end mode, all the reads are considered as the main ones, where as in the paired-end mode, we process the paired reads after the main reads. More specifically, when traversing trees, we firstly encode the main read s (as indicated by the node v) followed by matching and encoding of the paired read $\hat{s}$ (and followed by compressing the read identifier and base quality scores of both reads). The matching and encoding procedure of the paired read is as follows.

We maintain a buffer (a window) of $\hat{m}$ of previously encoded paired reads ($\hat{m} = 4096$ by default) with a simple index updated once a new read is added or an old one removed from the buffer. The reads in the buffer are indexed by their first (up to) 4 lexicographically smallest p-mers, which satisfy the same restrictions as signatures – we call them *restricted p-mers*.

Let $\hat{s}$ be the currently processed paired read. When searching for matches, we first retrieve a set of possibly matching reads $\tilde{s}$ which have been indexed by any of restricted p-mers found in $\hat{s}$. Then, we search in $\tilde{m}$ for the read which matches the best with $\hat{s}$, i.e., the distance $\hat{d}$ between the match $\tilde{s}$ and $\hat{s}$ is minimal. We analyze each potential match and calculate the distance $\hat{d}$ on the same basis as when performing matching of the main read (described in Subsection 1.1.3). The only difference is that the default value of the maximum distance threshold $\hat{d}_{\max}$ used in matching paired reads is set by default to $\hat{d}_{\max} = \frac{2}{3} \times \hat{r}$, where $\hat{r}$ is the length of the current paired read. Therefore, analogously, the read can be classified either as hard read (when no referential read was found in the buffer or the calculated distance $\hat{d} > \hat{d}_{\max}$), normal match ($\hat{d} \leq \hat{d}_{\max}$) or exact match ($\hat{d} = 0$).

The result of the reads matching is encoded in a number streams in an equivalent way as when encoding the result of the matching the main read, but with some minor differences. The streams which encode the matching result the same way are: 'Len-PE', 'Shift-PE', 'MatchRLE-PE', 'MatchBinary-PE', 'LettersX-PE'. The new ones or the ones which encode the information with minor differences are described below.

ReadFlags-PE. Stores the flag corresponding to a method of encoding the current read. The set of flags is the same as in case of 'ReadFlags' stream, excluding $f_{\text{read\_contig}}$. When encoding the paired read, we do not use any control flags (as when encoding the main read).

Swap-PE. Stores the binary flag telling whether during the reads distribution stage the second read from the pair has been swapped with the first one marking it as the main read (having the lexicographically smallest signature of both reads in the pair).

HardReads-PE. (Used only if 'ReadFlag-PE' is $f_{\text{read\_hard}}$). Here the hard reads are stored in a verbatim form – in contrast to "HardReads" stream used when encoding the main read, we do not skip storing the signature subsequence.

Prev-PE. (Used with all flags from 'ReadFlag-PE'). Stores the distance offset to the referenced read, i.e., the number of reads encoded between the reference read and the current one. The value of the offset is stored in 2 bytes.

Almost all the streams are compressed the same way as their equivalent ones used for encoding main reads. Only 'Prev-PE' stream is encoded using PPMd (and using the same parameters as for other streams). 'Swap-PE' is compressed using RC of order-4.

1.2 Compression of Quality Scores

FaStore provides two modes to compress and store the quality scores – lossless and lossy. In our work, we primarily focused on the latter, providing further compression (as compared to lossless) while being able to achieve variant calling results comparable – and sometimes superior – to those achieved with the original file (equivalently lossless compression). Thus in the following we focus on the lossy mode.

In FaStore we provide two general approaches to compress the quality scores in the lossy way – fixed and adaptive approaches. In the former, prior to compression, we apply a fixed data transformation to the quality scores, reducing the available resolution of the quality scores, which helps to reduce the size of the compressed stream. In the latter, we apply a controlled degree of information loss to the quality scores based on the statistics of the quality scores at hand. In other words, based on the observed quality scores, we try to select and apply the best scheme to compress it (based on user-specified requirements), reducing the size of the compressed quality scores stream while aiming to preserve the necessary information for the variant calling.

Independently of the approach used, the compressed quality scores are stored in a separate "Quality" stream. Next we explain in detail the fixed and adaptive schemes supported by FaStore.

Quality scores range	Mapped value
'N' (no call)	'N' (no call)
$2-9$	6
$10-19$	15
$20-24$	22
$25-29$	27
$30-34$	33
$35-39$	37
≥ 40	40

Table 1: Illumina 8-level quality values binning scheme [**?**]

1.2.1 Fixed schemes

In FaStore we provide two fixed quality scores transformations – Illumina 8-level binning [**?**] and binary thresholding. When selected, the transformation is applied during FASTQ reads distribution stage (see Subsection 1.1.2). Note that no prior information about the quality stream is required during the encoding step.

Illumina 8-level quality binning. When using this scheme, the resolution of quality scores is reduced to only 8 values, where the values of the quality scores are mapped according to Table 1. In case of "no call" ("N" symbol appearing at the corresponding DNA sequence position), we just do not store the quality score and replace it with "0" value when decompressing. To encode the quality scores, we apply range encoder of order-7, where the context is formed by the 6 previous symbols and a positional indicator $p = \frac{i}{8}$, where i is the position of the currently encoded base.

Binary thresholding. When using the binary thresholding scheme, the quality scores are classified either as "good" ($q_{\max}$) or "bad" ($q_{\min}$) according to a user-specified threshold parameter q_t, following the transformation:

$$q' = \begin{cases} q_{\min}, & \text{if } q < q_t \\ q_{\max}, & \text{otherwise} \end{cases} \tag{1}$$

By default, we use $q_{\max} = 40$, $q_{\min} = 6$ and $q_t = 20$. Similarly, as when performing Illumina 8-level binning, in case of "no call", we just do not store the quality score and replace it with "0" value when decompressing. To encode the quality scores, we apply range encoder of order-11, where the context is formed of the 10 previous symbols and a positional indicator $p = \frac{i}{2}$, where i is the position of the currently encoded base.

1.2.2 Adaptive approach

Our compressor uses QVZ [**?**] as its engine for adaptive quality scores compression. However, we introduce several improvements that result in better compression ratios for the same variant calling performance. Next we describe in detail the method used to compress the quality scores. As a side note, this scheme is also the used to compress the quality scores in a lossless manner.

Let N be the number of quality score sequences to be compressed. We denote each of these sequences as $\boldsymbol{Q}_j = [Q_{j,1}, Q_{j,2}, \ldots, Q_{j,L}]$, for $1 \leq j \leq N$. L denotes the length of the reads. We further denote the alphabet of the quality scores by $\mathcal{Q}$. For example, for *Phred + 33* scale, $\mathcal{Q} = \{33, 34, \ldots, 73\}$.

We model the quality score sequences by a Markov chain of order one, that is, for a given sequence $\boldsymbol{Q}$, we assume the probability that Q_i takes a particular value depends on previous values only through Q_{i-1}. We further assume that the quality score sequences are independent and identically (i.i.d.) distributed.

In brief, the steps to compress the quality scores are the following:

1. Compute the empirical transition probabilities of a Markov-1 Model from the data to be compressed. That is, $P(Q_i|Q_{i-1})$, for $1 \leq i \leq L$, with $P(Q_1|Q_0) = P(Q_1)$.
2. Based on the above computed probabilities, construct a codebook consisting on a set of quantizers. These quantizers are indexed by position (within the read) and the quantized value at the previous position (the context). They are constructed using a variant of the Lloyd-Max quantizer [**?**], which finds the optimal quantizers so as to satisfy a distortion constraint specified by the user.
3. Quantize the quality scores using the codebook constructed in the previous step and encode the results by means of an adaptive arithmetic encoder that achieves entropy-rate compression.

Next we explain each of these steps in more detail.

1.2.3 Computation of the empirical probabilities

Recall that the compression of the reads (DNA sequences) requires the data to be read as a pre-processing step. Thus we compute the desired probabilities during the pre-processing stage, avoiding having to read the data a second time.

1.2.4 Codebook generation

As stated above, the codebook consists of a set of quantizers indexed by position and quantized value at the previous position. Thus, for a given position $i \in \{1, 2, \ldots, L\}$, we construct as many quantizers as unique quantized values at position $i-1$ across all quality score sequences. For a given quality score Q, we denote its quantized value as $\hat{Q}$, such that the quantized

quality score sequences are represented as $\hat{\boldsymbol{Q}}_j = [\hat{Q}_{j,1}, \hat{Q}_{j,2}, \ldots, \hat{Q}_{j,L}]$, with $1 \leq j \leq N$. Next we explain how the quantizers are constructed.

Given a random variable Q governed by the probability mass function $P(\cdot)$ over the alphabet $\mathcal{Q}$ of size K, let $\mathbf{D} \in \mathbb{R}^{K \times K}$ be a distortion matrix where each entry $D_{q,\hat{q}} = d(q, \hat{q})$ is the penalty for reconstructing symbol q as $\hat{q}$. We further define $\hat{\mathcal{Q}} \subseteq \mathcal{Q}$ to be the alphabet of the quantized values of size $M \leq K$.

The quantizer, denoted hereafter as $LM(\cdot)$, is a mapping $\mathcal{Q} \to \hat{\mathcal{Q}}$ that minimizes the expected distortion. Specifically, the quantizer seeks to find a collection of boundary points $b_k \in K$ and reconstruction points $\hat{q}_k \in M$, where $k \in \{1, 2, \ldots, M\}$, such that the quantized value of symbols $q \in \mathcal{Q}$ is given by the reconstruction point of the region to which it belongs. That is, the quantizer aims to minimize

$$\{b_k, \hat{q}_k\}_{k=1}^{M} = \underset{b_k, \hat{q}_k}{\operatorname{argmin}} \quad \sum_{j=1}^{M} \sum_{q=b_{j-1}}^{b_j - 1} P(q) d(q, \hat{q}_j). \tag{2}$$

In order to solve this problem we perform a one-dimensional weighted k-means algorithm, where after initializing the boundary points b_k, the algorithm iteratively performs as follows: i) for each region k choose the $\hat{q}_k \in \{b_{k-1}, \ldots, b_k - 1\}$ that minimizes $\sum_{q=b_{k-1}}^{b_k - 1} P(q) d(q, \hat{q})$, and ii) assign each point q to the closest reconstructed point $\hat{q}_k$, where the distance is measured as $d(q, \hat{q})$, yielding new boundary points b_k. The algorithm stops if no further change is obtained in the b_k or after a fixed number of iterations.

Given a distortion matrix $\mathbf{D}$, the defined quantizer depends on the number of regions M and the input probability mass function $P(\cdot)$. Thus we denote the quantizer with M regions based on probability mass function $P(\cdot)$ as $LM_M^P(\cdot)$, and the quantized value of a symbol $q \in \mathcal{Q}$ as $LM_M^P(q)$.

Note that a reconstructed point $\hat{q}$ has probability of occurrence $P(\hat{q}) = \sum_{q: LM_M^P(q) = \hat{q}} P(q)$. Thus, each generated quantizer $LM_M^P(\cdot)$ defines a rate-distortion pair, where the rate and distortion are given by

$$R(LM_M^P(\cdot)) = \sum_{\hat{q} \in \hat{\mathcal{Q}}} P(\hat{q}) \log_2 P(\hat{q}) \quad \text{and} \quad D(LM_M^P(\cdot)) = \sum_{\hat{q} \in \hat{\mathcal{Q}}} \sum_{q: LM_M^P(q) = \hat{q}} P(\hat{q}) d(q, \hat{q}),$$

respectively. Furthermore, for a fixed probability mass function $P(\cdot)$, the only varying parameter is the number of regions M. Since M needs to be an integer, not all rate-distortion pairs are achievable. Thus, as done in QVZ [**?**], we define an extended version of the LM quantizer, which consists of two LM quantizers with the number of regions given by ρ and $\rho + 1$, each of them used with probability $1 - r$ and r, respectively (where $0 \leq r \leq 1$).

However, in contrast to QVZ, which aims at achieving an arbitrary rate (same for all quantizers), we aim at achieving an arbitrary distortion D (we discuss below the reason for this choice, which was also made in [**?**]). Therefore, ρ is given by the maximum number of regions such that $D(LM_\rho^P(Q)) > D$ (which implies $D(LM_{\rho+1}^P(Q)) < D$). Then, the probability r is chosen such that the average distortion is equal to D.

The reason for setting all quantizers to the same distortion D is the following. Given that there are at most $L \times K$ quantizers (indexed by position and previously quantized value), the final rate R is given by the convex combination of the individual rates R_i of all the quantizers. Thus, one can pose the following optimization problem:

$$\begin{aligned} \underset{R_i}{\text{minimize}} \quad & \sum_i \alpha_i Ri \\ \text{subject to} \quad & \sum_i \alpha_i K_i \exp(-h_i R_i) = D, \end{aligned}$$

where we have assumed that the rate-distortion function generated by each of the quantizers is of the form $D_i(R_i) = K_i \exp(-h_i R_i)$ [**?**].

Solving this problem using the Lagrange multipliers method, we obtain that the optimal distortion at which each quantizer must operate is given by

$$D_i = \frac{D}{h_i \sum_i \frac{\alpha_i}{h_i}}.$$

For the case under consideration, h_i may not be computable in some cases. Moreover, we expect all quantizers to exhibit a similar behavior. Thus, we assume $h_i = h \ \forall i$, which translates into all quantizers targeting the same distortion D. Note that a distortion D equal to zero corresponds to lossless compression.

In order to compute the quantizers defined above, we need to specify the input probability $P(\cdot)$. For the case under consideration, this probability is given by $P(Q_i|\hat{Q}_{i-1})$, where i denotes the position within the read (i.e., $1 \leq i \leq L$). These probabilities can be computed from the empirically computed probabilities $P(Q_i|Q_{i-1})$ $(1 \leq i \leq L)$. We refer the reader to [**?**] for the exact derivation.

We denote the set of quantizers as $\{\mathcal{Q}^i_{\hat{q}}\}$, where i denotes the position, and $\hat{q}$ the value in the previous context $i-1$. The codebook generation is summarized in Algorithm 2.

1.2.5 Encoding

The encoding process is performed read by read. For each read, we quantize all quality scores sequentially, starting at the first position. The latest quantized value serves as context to quantize the quality score at the next position. As the quantization takes place, the quantized values are passed to an adaptive arithmetic encoder, which uses a separate model for each position and context.

1.2.6 Note on the distortion metric

By default, FASTORE aims at minimizing the Mean Squared Error (MSE) distortion between the original and quantized quality scores. However, the considered adaptive lossy compressor for the quality scores works with any quasi-convex distortion metric. The user can provide it as input to the program in the form of a matrix. As a reference, Table 2 lists some of the most

Algorithm 2 Codebook generation

Input: Transition probabilities $P(Q_i \mid Q_{i-1})$, distortion value D
Output: Codebook: collection of quantizers $\{\mathcal{Q}^i_{\hat{q}}\}$
$P \leftarrow P(Q_1)$
Compute and store $\mathcal{Q}^1$ based on P and the corresponding ρ so as to achieve desired distortion D
for all columns $i = 2$ to L **do**
  Compute $P(Q_{i-1} \mid \hat{Q}_{i-1} = \hat{q}),\ \forall\ \hat{q} \in \text{support}(\hat{Q}_{i-1})$
  for all $\hat{q} \in \text{support}(\hat{Q}_{i-1})$ **do**
    $P \leftarrow P(Q_i \mid \hat{Q}_{i-1} = \hat{q})$
    Compute and store $\mathcal{Q}^i_{\hat{q}}$ based on P and the corresponding ρ so as to achieve desired distortion D
  end for
end for

Table 2: Distortion metrics used for assessment of the lossy compressors in terms of rate-distortion performance. Q is the original quality score and $\widehat{Q}$ is the reconstructed one after lossy compression.

MSE	$d_{mse}(Q,\widehat{Q}) = \frac{1}{L\cdot N}\sum_{j=1}^{N}\sum_{i=1}^{L}\lvert Q_{j,i}-\widehat{Q}_{j,i}\rvert^2$
L1	$d_{\ell 1}(Q,\widehat{Q}) = \frac{1}{L\cdot N}\sum_{j=1}^{N}\sum_{i=1}^{L}\lvert Q_{j,i}-\widehat{Q}_{j,i}\rvert$
Lorentzian	$d_L(Q,\widehat{Q}) = \frac{1}{L\cdot N}\sum_{j=1}^{N}\sum_{i=1}^{L}\log(1+\lvert Q_{j,i}-\widehat{Q}_{j,i}\rvert)$
Chebyshev	$d_C(Q,\widehat{Q}) = \frac{1}{N}\sum_{j=1}^{N}\max_{1\le i\le L}\lvert Q_{j,i}-\widehat{Q}_{j,i}\rvert)$
Max-Min	$d_{MM}(Q,\widehat{Q}) = \frac{1}{N}\sum_{j=1}^{N}\max_{1\le i\le L}\frac{\max(Q_{j,i},\widehat{Q}_{j,i})}{\min(Q_{j,i},\widehat{Q}_{j,i})}$
Soergel	$d_S(Q,\widehat{Q}) = \frac{1}{N}\sum_{j=1}^{N}\frac{\sum_{i=1}^{L}\lvert Q_{j,i}-\widehat{Q}_{j,i}\rvert}{\sum_{i=1}^{L}\max(Q_{j,i},\widehat{Q}_{j,i})}$

widely used distortion metrics in practice. MSE has been chosen as the default distortion as it provides the best results [**?**, **?**].

1.3 Compression of Read Identifiers

The FASTQ read identifiers start with the '@' character and are followed by an arbitrary sequence identifier. Optionally, they can also contain a comment, represented as a content appearing after the first whitespace character. The identifier is a free text format with no limit on its length [**?**] and which can can hold information such as instrument name, unique read number, flowcell lane, read length, etc. Usually, for the FASTQ reads which have been generated from the same sequencing experiment and using the same instrument, the information stored in the read identifiers follow the same convention. Therefore, the read identifier can be perceived as a concatenation of different fields (tokens) separated by a set of delimiters, where each of the tokens can be of a character, string or numeric type.

Moreover, if the FASTQ reads have been generated from a sequencing library created in paired-end configuration, the read identifiers of a pair of reads will share a significant portion of the content. They will possibly differ on the tokens indicating the number of the read in the pair (e.g., "1" or "2"). Therefore, in FaStore, when processing reads in paired-end mode, we store only one read identifier per pair of reads (and keep the information about possible differences), which allows to reduce the size of the compressed read identifiers up to half of what they would occupy in single-end compression mode.

In FaStore, we provide an option to compress the read identifiers in lossless (flag '-H') or lossy modes, with the focus on the former one. In lossy mode, we can either skip storing the comments (flag '-C') or completely skip storing the identifiers (the default option), generating a unique identifier per read or pair of reads while decompressing.

In brief, depending on the selected read identifiers compression mode, the steps to compress them are the following:

1. Tokenize all the read identifiers gathering statistics for each token.
2. Based on the gathered statistics compress the identifiers, encoding each token separately.

1.3.1 Tokenization

During the analysis and tokenization of the read identifiers, we keep a (global) set of statistics, indexed by number of the currently processed token of the identifier. We process the identifiers as follows. We split the identifier into a set of tokens t_i using a set of delimiters (such as comma, dot, semicolon, etc.). We use a set of dictionaries D_i (where D_i correspond to the i-th token) and a set of pairs $v_i^{\min}, v_i^{\max}$ to assess the type of the token and its possible values. For each t_i, we check its type and compare its value with the previous ones stored in D_i. However, we only keep a track of maximum 255 distinct values. Additionally, if t_i is of numeric type, we also update the observed minimum $v_i^{\min}$ and maximum $v_i^{\max}$ values.

When processing reads in single-end mode, each read with its identifier is treated separately. However, in paired-end mode, as the significant part of the read identifier is shared between both reads, we analyze them together to check which are the tokens that they possibly differ by. As a side note, we perform the tokenization and gathering of statistics during reads distribution stage (see Subsection 1.1.2) and we compress the identifiers in parallel while compressing the DNA sequence and base quality scores.

1.3.2 Encoding

Having calculated the global statistics for the tokens, we can proceed to encode them. We use two streams to store them, namely 'Idx' and 'Val'. Let t_i be the i-th token of the current read, which is being encoded. If t_i is a fixed token, i.e., only one possible value was observed in D_i, it is being skipped. Otherwise, if the token is of a string type and the number of its possible distinct values is less than 255, we obtain the index idx of the value of t_i stored in D_i. Then, we encode idx value in 'Idx' stream using the number i of the token as a context. Finally,

the token of character, numerical or string type, but for which the number of distinct values is greater than 255, is directly encoded in 'Val' stream using i as a context. The numerical values, however, are previously rescaled, subtracting the minimal observed value $\mathrm{v\_min}_i$ from them. In case of paired-end compression, only one read identifier is encoded per one pair of reads. Encoding the variable part of the identifier shared between two reads from the pair (denoting whether the read is either the first or the second read from the pair) is already done while encoding the pairing information between two DNA sequences (in 'Swap-PE' streams).

The streams are finally compressed using our range coder of order-2, using one previous symbol and the number i of the currently encoded token as a context. At the end of the compression, the set of dictionaries D_i and with the set of $\mathrm{v\_min}_i, \mathrm{v\_max}_i$ values (but only for the numerical tokens) is stored in the file footer. Moreover, for paired-end compression, we also store in the footer the corresponding numbers of the variable fields in the paired reads.

2 Datasets

This section describes the datasets used in the experiments and reports the links used to download them.

HS2 dataset

This dataset was previously used in ORCOM [**?**]. It consists of 48 compressed FASTQ files, which can be downloaded from:

ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024163/ERR024163_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024163/ERR024163_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024164/ERR024164_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024164/ERR024164_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024165/ERR024165_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024165/ERR024165_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024166/ERR024166_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024166/ERR024166_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024167/ERR024167_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024167/ERR024167_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024168/ERR024168_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024168/ERR024168_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024169/ERR024169_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024169/ERR024169_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024170/ERR024170_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024170/ERR024170_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024171/ERR024171_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024171/ERR024171_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024172/ERR024172_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024172/ERR024172_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024173/ERR024173_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024173/ERR024173_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024174/ERR024174_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024174/ERR024174_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024175/ERR024175_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024175/ERR024175_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024176/ERR024176_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024176/ERR024176_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024177/ERR024177_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024177/ERR024177_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024178/ERR024178_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024178/ERR024178_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024179/ERR024179_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024179/ERR024179_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024180/ERR024180_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024180/ERR024180_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024181/ERR024181_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024181/ERR024181_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024182/ERR024182_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024182/ERR024182_2.fastq.gz

ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024183/ERR024183_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024183/ERR024183_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024184/ERR024184_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024184/ERR024184_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024185/ERR024185_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024185/ERR024185_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024186/ERR024186_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR024/ERR024186/ERR024186_2.fastq.gz

The downloaded files were decompressed, concatenated pairwise, truncating all reads to 100 bp, which can be done as follows:

```
gunzip -c ERR0241*_1.fastq.gz | cut -c1-100 >> HS2_1.fastq
gunzip -c ERR0241*_2.fastq.gz | cut -c1-100 >> HS2_2.fastq
```

The result was stored as two FASTQ files: HS2_1.fastq and HS2_2.fastq.

GG dataset

This dataset was previously used in ORCOM [**?**]. It consists of 12 compressed FASTQ files, which can be downloaded from:

ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030308/SRX043656/SRR105788_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030308/SRX043656/SRR105788_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030309/SRX043656/SRR105789_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030309/SRX043656/SRR105789_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030312/SRX043656/SRR105792_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030312/SRX043656/SRR105792_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030314/SRX043656/SRR105794_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA030/SRA030314/SRX043656/SRR105794_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036382/SRX043656/SRR197985_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036382/SRX043656/SRR197985_2.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036383/SRX043656/SRR197986_1.fastq.bz2
ftp://ftp.ddbj.nig.ac.jp/ddbj_database/dra/fastq/SRA036/SRA036383/SRX043656/SRR197986_2.fastq.bz2

The downloaded files were decompressed and concatenated pairwise.The result was stored as two FASTQ files: GG_1.fastq and GG_2.fastq.

HSX dataset

HSX dataset consists of 2 FASTQ files, which compressed can be downloaded from:

https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12878D_HiSeqX_R1.fastq.gz
https://dnanexus-rnd.s3.amazonaws.com/NA12878-xten/reads/NA12878D_HiSeqX_R2.fastq.gz

The downloaded files were decompressed and stored as HSX_1.fastq and HSX_2.fastq.

WGS-14, WGS-42, and WGS-235x datasets

WGS-235 dataset consists of 36 FASTQ files (18 pairs) coming from Illumina Platinum Genomes [**?**]. They can be downloaded in a compressed form from:

ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174324/ERR174324_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174324/ERR174324_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174325/ERR174325_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174325/ERR174325_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174326/ERR174326_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174326/ERR174326_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174327/ERR174327_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174327/ERR174327_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174328/ERR174328_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174328/ERR174328_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174329/ERR174329_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174329/ERR174329_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174330/ERR174330_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174330/ERR174330_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174331/ERR174331_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174331/ERR174331_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174332/ERR174332_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174332/ERR174332_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174333/ERR174333_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174333/ERR174333_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174334/ERR174334_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174334/ERR174334_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174335/ERR174335_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174335/ERR174335_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174336/ERR174336_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174336/ERR174336_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174337/ERR174337_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174337/ERR174337_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174338/ERR174338_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174338/ERR174338_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174339/ERR174339_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174339/ERR174339_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174340/ERR174340_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174340/ERR174340_2.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174341/ERR174341_1.fastq.gz

ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR174/ERR174341/ERR174341_2.fastq.gz

WGS-14 and WGS-42 datasets are subsets of WGS-235 – WGS-14 consists of only first pair of files (ERR174324_1.fastq.gz and ERR174324_2.fastq.gz) and WGS-42 of first 3 pairs. The files corresponding to WGS-14 and WGS-42 datasets were decompressed and concatenated pairwise, resulting in WGS_14_1.fastq and WGS_14_2.fastq and WGS_42_1.fastq and WGS_42_2.fastq files, respectively.

Analogously, when analyzing how the compression ratio changes with the sequencing coverage, we sampled the WGS-235 dataset into 1, 2, 3, 4, 6, 9, 12, and 18 pairs of FASTQ files, respectively.

CE dataset

CE dataset consists of 2 FASTQ files used previously in LEON [**?**]. They can be downloaded in a compressed form from:

ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR065/SRR065390/SRR065390_1.fastq.gz
ftp://ftp.sra.ebi.ac.uk/vol1/fastq/SRR065/SRR065390/SRR065390_2.fastq.gz

The downloaded files were decompressed and stored as CE_1.fastq and CE_2.fastq.

WEX dataset

WEX dataset is available as aligned data in BAM format coming from GIAB [**?**]. It can be downloaded from:

ftp://ftp-trace.ncbi.nlm.nih.gov/giab/ftp/data/AshkenazimTrio/HG002_NA24385_son/OsloUniversityHospital_Exome/151002_7001448_0359_AC7F6GANXX_Sample_HG002-EEogPU_v02-KIT-Av5_AGATGTAC_L008.posiSrt.markDup.bam

After downloading the file it was renamed to WEX.bam. Next, we used SAMTools [**?**] (version 1.2) to transcode it to FASTQ format:

```
# sort the file by read names before transcoding
samtools sort -n WEX.bam > WEX-sorted.bam

# transcode to FASTQ
samtools fastq -1 WEX_1.fastq -2 WEX_2.fastq -0 WEX.fastq WEX.bam
```

The result of transcoding is stored as 2 files – WEX_1.fastq and WEX_2.fastq. There are no unpaired reads – in that case, they would had been stored in a separate WEX.fastq file.

Metagenomic datasets

Metagenomic dataset were downloaded from the following sites:

https://www.ebi.ac.uk/ena/data/view/SRR359032
https://www.ebi.ac.uk/ena/data/view/ERR532393
https://www.ebi.ac.uk/ena/data/view/ERR1474585

3 Compression test – tools invocation

The applications were tested in multithreaded mode (if supported) using 8 threads. In the case that the application did not provide an option to compress the FASTQ files in paired-end mode (or provides explicitly a single-end mode), the two paired files were concatenated into one large FASTQ file prior to compression. Application execution times were measured using Linux command "time".

The final tests were performed on the CNAG cluster. It is made of more than 100 compute nodes each one having two Intel Xeon E52670 2.60 GHz processors with 128 GB of RAM. It has about 3 PB of network-distributed hard-drive storage mounted as a Lustre parallel file system (`http://lustre.org/`). Inter-node communication is performed via a dedicated Infiniband network, whereas the Lustre filesystem is connected to the cluster via a number of standard Gigabit Ethernet connections. Each single performance test was performed on a fully reserved node to avoid possible interference with other applications.

3.1 Compression of FASTQ files

PIGZ (gzip) We tested PIGZ (a parallel version of gzip) in version 2.3.3.

- To compress:
  ```
  pigz -9 -p 8 -c IN.fastq > COMP.gz
  ```
- To decompress:
  ```
  pigz -d -p 8 -c COMP.gz > OUT.fastq
  ```

DSRC2 We tested DSRC2 [**?**] in version 2.1.0, which was downloaded from `https://github.com/lrog/dsrc`.

- To compress using *DSRC-FAST*:
  ```
  dsrc c -m0 -t8 -v IN.fastq COMP.dsrc
  ```
- To compress using *DSRC-MAX*:
  ```
  dsrc c -m2 -t8 -v IN.fastq COMP.dsrc
  ```
- To decompress:
  ```
  dsrc d -t8 COMP.dsrc OUT.fastq
  ```

FQZCOMP We tested FQZCOMP [**?**] in version 4.6, which was downloaded from `https://sourceforge.net/projects/fqzcomp/`.

- To compress using *FQZCOMP-STD*:
  ```
  fqz_comp IN.fastq COMP.fqz
  ```
- To compress using *FQZCOMP-MAX*:
  ```
  fqz_comp -n2 -q3 -s8+ -b IN.fastq COMP.fqz
  ```
- To decompress:
  ```
  fqz_comp -d COMP.fqz OUT.fastq
  ```

QUIP We tested QUIP [?] in version 1.1.8, which was downloaded from `https://github.com/dcjones/quip`.

- To compress using *QUIP-FQ-STD*:
  ```
  quip -v -c IN.fastq > COMP.qp
  ```
- To compress using *QUIP-FQ-MAX*:
  ```
  quip -a -v -c IN.fastq > COMP.qp
  ```
- To decompress:
  ```
  quip -d -c COMP.qp OUT.fastq
  ```

SCALCE We tested SCALCE [?] in version 2.8, which was downloaded from `http://sfu-compbio.github.io/scalce/`.

- To compress using *SCALCE-SE*:
  ```
  scalce -T 8 -o COMP.sc IN.fastq
  ```
- To decompress using *SCALCE-SE*:
  ```
  scalce -d -T 8 -o OUT.fastq COMP.sc_1.scalcen
  ```
- To compress using *SCALCE-PE*:
  ```
  scalce -T 8 -r -o COMP.sc IN.fastq
  ```
- To decompress using *SCALCE-PE*:
  ```
  scalce -d -r -T 8 -o OUT.fastq COMP.sc_1.scalcen
  ```

LEON We tested LEON [?] in version 1.0.0, which was downloaded from `http://gatb.inria.fr/software/leon/`.

- To compress:
  ```
  leon -c -file IN.fastq -nb-cores 8 -lossless
  ```
- To decompress:
  ```
  leon -d -file IN.leon -nb-cores 8
  ```

3.2 Compression of DNA sequences

ORCOM We tested ORCOM [?] in version 1.0rc, which was downloaded from `https://github.com/lrog/orcom/`.

- To compress:
  ```
  orcom_bin  e -t8 -v -iIN.fastq -oCOMP.bin
  orcom_pack e -t8 -v -iCOMP.bin -oCOMP.pack
  ```
- To decompress:
  ```
  orcom_pack d -t8 -iCOMP.pack -oOUT.dna
  ```

MINCE We tested MINCE [?] in version 0.6.1, which was downloaded from `http://www.cs.cmu.edu/ckingsf/software/mince`.

- To compress in single-end mode:

```
mince -e -r IN.fastq -o COMP -p 8
```

- To compress in paired-end mode:

```
mince -e -1 IN_1.fastq -2 IN_2.fastq -o COMP -p 8
```

- To decompress:

```
mince -d -i COMP -o OUT.dna -p 8
```

BEETL We tested BEETL [**?**] in version 1.1.0, which was downloaded from `https://github.com/BEETL/BEETL`.

- To compress:

```
# run BEETL-BWT
beetl-bwt -i IN.fastq -o COMP.beetl --output-format ASCII \
   --sap-ordering --algorithm ext

# compress the output BWT runs uzing 7za with PPMd
7za a -mmt=$2 -mm=PPMd -mmem=256m -mo=4 COMP-B.zip $TMP_PFX.beetl-B*
7za a -mmt=$2 -mm=PPMd -mmem=256m -mo=4 COMP-P.zip $TMP_PFX.beetl-P*
7za a -mmt=$2 -mm=PPMd -mmem=256m -mo=4 COMP-S.zip $TMP_PFX.beetl-S*
```

- To decompress:

```
# decompress the BWT runs
7za e -mmt=$2 COMP-B.zip
7za e -mmt=$2 COMP-P.zip
7za e -mmt=$2 COMP-S.zip

# run BEETL-UNBWT
beetl-unbwt -i COMP.beetl -o OUT.fasta --output-format fasta
```

3.3 Running FaStore

FaStore offers a variety of different compression configurations. Hence, for an easier selection, we created 4 profiles, namely *lossless*, *reduced*, *lossy* and *max*. To perform automatic compression and decompression, a pair of scripts 'fastore_compress.sh' and 'fastore_decompress.sh' is available in the FaStore package.

- To compress files in paired-end mode:

```
bash fastore_compress.sh --in IN_1.fastq --pair IN_2.fastq --out COMP \
    --threads 8 PROF [--fast] [--signature p]
```

- To decompress in paired-end mode:

```
bash fastore_decompress.sh --in COMP --out OUT_1.fastq \
    --pair OUT_2.fastq --threads 8
```

- To compress file in single-end mode:

```
bash fastore_compress.sh --in IN.fastq --out COMP \
    --threads 8 PROF [--fast] [--signature p]
```

- To decompress in single-end mode:

```
bash fastore_decompress.sh --in COMP --out OUT.fastq --threads 8
```

In all cases, the *PROF* parameter specifies the profile, which can be: '–lossless', '–reduced', '–lossy' or '–max'. Alternatively, '–fast' switch can be provided which will launch FaStore in *C0* DNA compression mode (*C1* by default). When compressing, FaStore uses a signature of length $p = 8$, however, '–signature p' switch can be used to specify a different one. The compressed files will be stored as COMP.cmeta and COMP.cdata files.

4 Variant calling test – tools invocation

The analysis of the possible impact of performing lossy quality scores compression and reordering the reads on variant calling was performed on WGS-14 and WGS-42 datasets, both generated from paired-end library. For the analysis, we run multiple singular experiments, each consisting of two steps:

- preprocessing of the input FASTQ files,
- performing mapping and variant calling.

As a point of reference, we performed mapping and variant calling using the original files.

4.1 Preprocessing of FASTQ files

4.1.1 Reordering the reads in original FASTQ files

As the input datasets were generated from paired-end library, each stored as a pair of FASTQ files, after reordering the reads, it is crucial to preserve the pairing between the reads in resulting files on the file level (i.e., a pair of reads resides on the same lines in both files). For reordering we used the script 'shuffle_sort.sh' available in the FaStore package. It can be run as:

```
bash shuffe_sort.sh IN_1.fastq IN_2.fastq
```

the output of reordering the reads inside files will be stored as shuf-IN_1.fastq and shuf-IN_2.fastq.

4.1.2 Transforming quality scores in original FASTQ files

We tested three different transformations of base quality scores performed on original FASTQ files: Illumina 8-level binning [**?**], binary thresholding and quantization based on QVZ [**?**]. The algorithms have been explained in details in Section 1.2.

Illumina 8-level binning and binary thresholding transformations were implemented in 'downsample_quality.py' script available in the FaStore package, which can be run as:

```
python downsample_fastq.py IN.fastq OUT.fastq MODE [THR]
```

where *MODE* specifies the transformation ('B' for Illumina binning, 'T' for thresholding) and *THR* specifies the threshold value used with the former transformation. The script needs to be run for both FASTQ files separately.

The quantization of quality scores was performed using QVZ standalone application, which can be downloaded from `https://github.com/mikelhernaez/qvz2`. To perform quantization, we use the script 'quantize_qvz.sh' available in the FaStore package, which can be run as:

```
bash quantize_qvz.sh IN.fastq DIST OUT.fastq
```

where *DIST* specifies the distortion value. We refer to Section 1.2.6 for the different distortions allowed by QVZ. The script needs to be run for both FASTQ files separately.

Previous to running the script, the user needs to use the C file 'replace_qual_fastq.c' also available in the FaStore package, and compile it as follows:

```
gcc -o replace_qual_fastq replace_qual_fastq.c
```

Finally, the user needs to modify the paths to QVZ and the executable C file in the quantize_qvz.sh script.

4.1.3 Preprocessing FASTQ files using FaStore

We run FaStore in 'C0' mode testing different lossy quality compression modes alongside lossless. We run FaStore as a set of commands:

```
# 1. bin the reads
fastore_bin e -i"IN_1.fastq IN_2.fastq" -o"__tmp.bin" \
    -p8 -s10 -H -z -tTH QUA

# 2. compress the reads
fastore_pack e -i"__tmp.bin" -o"__tmp.pack" \
    -f256 -c10 -d8 -w256 -W256 -z -tTH

# 3. decompress the reads
fastore_pack d -i"__tmp.pack" -o"OUT_1.fastq OUT_2.fastq" -z -tTH
```

where *TH* specifies the number of processing threads and *QUA* specifies the quality compression option, which can be:

- '-q0' – lossless mode
- '-q1' – binary thresholding
- '-q2' – Illumina 8-level binning
- '-q3 -Dn' – QVZ mode using 'n' distortion level.

It's important to note, that the options to compress quality scores alongside read identifiers need to be specified during the binning stage.

4.2 Variant calling

To obtain the analysis-ready set of variants, we follow steps as recommended by Genome Analysis Toolkit (GATK) [**?**] Best Practices [**?**]. We use BWA-MEM [**?**, **?**] in version 0.7.10 to map the reads to the human reference genome version GRCh37. We convert the resulting SAM flies into BAM format and sort by chromosome and position using SAMTools [**?**] in version 1.2. Then, we mark duplicates, add/replace read groups, and index the BAM file using Picard tools (`https://broadinstitute.github.io/picard/`) in version 2.4.1. Following, we apply GATK Base Quality Score Recalibration (BQSR) and perform variant calling

using GATK HaplotypeCaller both for SNPs and INDELs. For completeness, we optionally perform filtering of the variants using GATK Variant Quality Scores Recalibration (VQSR). We compare the variants (both filtered and not) using as a "gold standard" the set of variants from the Genome In A Bottle (GIAB) consortium [**?**] in version 3.2.2. Finally, we compare our results using Haplotype Comparison Tools provided by Illumina (available at `https://github.com/Illumina/hap.py`), which is also recommended by Global Alliance for Genomics and Health (GA4GH) as one of benchmarking standards. The pipeline is available in the FaStore package as 'GATK_BestPractices_pipe_happy_NIST.sh'.

Before running the pipeline, it is necessary to set the paths to the binaries and data resources (e.g., reference sequence, picard tools, etc.) in the provided script. The user also needs to provide some input arguments like the path to the FASTQ files under consideration. In particular, the script can be run as follows:

```
bash GATK_BestPractices_pipe_happy_NIST.sh temp_location_path num_threads \
    IN_1.fastq IN_2.fastq
```

5 Additional results

5.1 Comparison of "normal" vs. "fast" modes

Table 3: Trade-off between modes C0 (fast) and C1 (default).

	Compr. factor of DNA stream			Compr. speed		
Data set	C0 [%]	C1 [%]	Ratio C0/C1	C0 [MB/s]	C1 [MB/s]	Ratio C0/C1
CE	6.96	6.35	1.09	35.5	3.4	10.56
GG	8.96	8.41	1.07	39.2	6.2	6.30
HS2	5.88	5.34	1.10	11.9	6.6	1.82
HSX	8.91	8.03	1.11	32.2	9.1	3.52
WEX	5.60	5.38	1.04	35.1	5.2	6.74
WGS-14	10.82	10.28	1.05	32.5	5.1	6.39
WGS-42	7.14	6.47	1.10	29.7	8.5	3.50

5.2 FaStore resources usage

Table 4: Example of temporary peak disk and RAM usage when compressing the WGS-14 and WGS-42 datasets. "Fast" refers to C0 mode (in contrast to default mode C1).

	Peak HDD [GB]		Peak RAM [GB]	
Data set	WGS-14	WGS-42	WGS-14	WGS-42
Lossless fast	73	208	19	52
Lossless	108	314	17	47
Reduced fast	42	118	15	38
Reduced	62	181	14	39
Lossy fast	59	165	15	40
Lossy	95	278	16	43
Max fast	22	59	12	34
Max	33	95	12	34

5.3 Metagenomics results

Prior to compression, the metagenomics datasets have been preprocessed, by downsampling the quality scores (Illumina binning) and trimming the read identifiers (for compatibility with SCALCE as it trims them by default). When running FaStore, we reduced the length of the signature to $p = 6$ symbols. When compressing small datasets it may be better to use a shorter signature length, as it defines the number of available bins into which the reads will be distributed. Compressing bins containing small number of reads will result in degraded compression efficiency.

Table 5: Example metagenomics dataset – compression results reported for whole archives. Sizes are reported in gigabytes. Ratio is expressed as the size of the raw FASTQ file divided by it's compressed size.

Dataset	File	Raw	gzip		DSRC2		Quip		FQZcomp		LEON		SCALCE		FaStore	
		Size	Size	Ratio	Size	Ratio	Size	Ratio	Size	Ratio	Size	Ratio	Size	Ratio	Size	Ratio
ERR1662204	_1	0.96	0.24	4.028	0.16	6.075	0.16	6.148	0.15	6.481	0.19	5.176	0.14	6.643	0.13	7.587
	_2	0.96	0.24	3.997	0.16	6.029	0.16	6.101	0.15	6.414	0.19	5.127	0.15	6.559	0.13	7.451
ERR532393	_1	8.00	2.22	3.601	1.47	5.447	1.42	5.625	1.07	7.499	1.23	6.476	1.19	6.696	0.91	8.752
	_2	8.00	2.26	3.541	1.49	5.350	1.45	5.520	1.14	7.019	1.33	6.032	1.29	6.196	1.01	7.913
SRR359032	_1	3.87	0.97	4.012	0.61	6.385	0.46	8.408	0.33	11.616	0.42	9.168	0.41	9.552	0.31	12.341
	_2	3.87	0.98	3.957	0.62	6.299	0.46	8.344	0.34	11.510	0.43	9.079	0.41	9.491	0.32	12.130

Table 6: Example metagenomics dataset – compression results reported for particular data streams. Sizes are reported in megabytes.

Dataset	File	DSRC2			Quip			FQZcomp			LEON			SCALCE			FaStore		
		ID	DNA	QUA	ID	DNA	QUA	ID	DNA	QUA	ID	DNA	QUA	ID	DNA	QUA	ID	DNA	QUA
ERR1662204	_1	0	106	52	0	105	51	0	98	50	0	110	75	17	67	60	12	59	55
	_2	0	106	53	0	105	52	0	98	51	0	110	77	17	68	61	12	60	57
ERR532393	_1	0	883	585	0	839	582	0	491	576	0	407	828	172	368	654	116	205	593
	_2	0	881	614	0	840	609	0	533	606	0	463	862	173	429	689	116	266	629
SRR359032	_1	0	397	210	0	248	212	0	127	207	0	112	310	81	89	235	54	43	217
	_2	0	396	219	0	243	222	0	121	216	0	105	322	81	84	243	54	38	227

5.4 Variant calling results with applied filtering

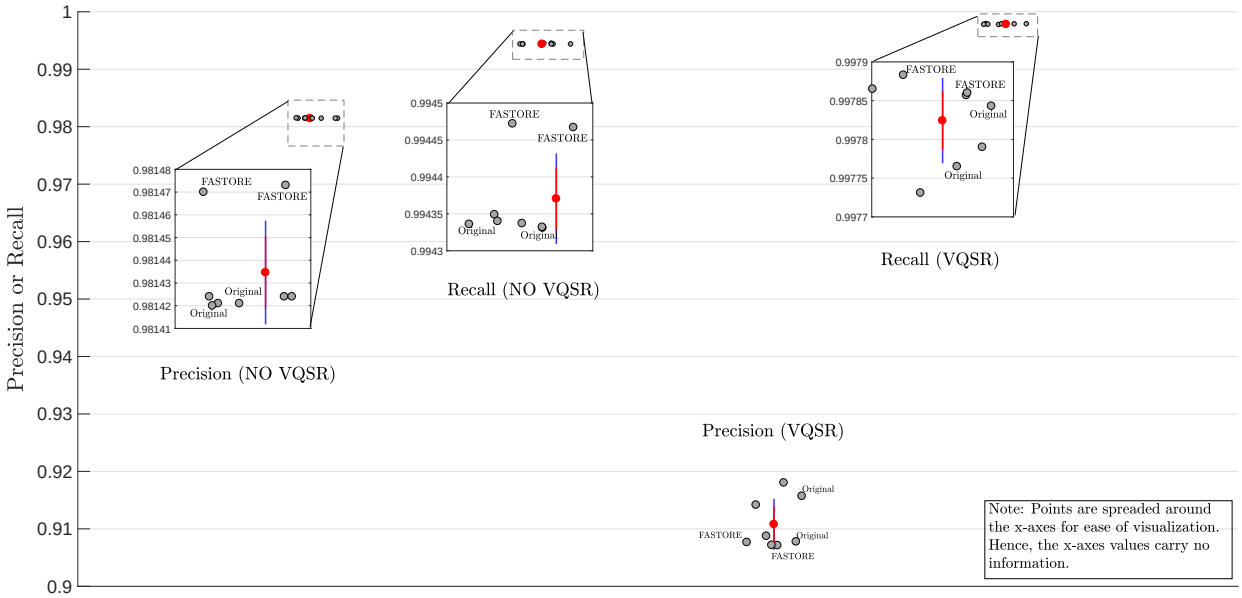

Figure 1: Precision and Recall obtained for various reorderings of dataset WGS-14, with and without VQSR filtering. Points without label correspond to random shuffling. The red point represents the mean, the red line is the 95% confidence on the mean, and the blue line is the standard deviation. The y-axis represents precision or recall, based on what it is specified in the x-axis.