ParRefCom : Parallel Reference-based Compression of Paired-end Genomics Read Datasets

Nagakishore Jammula
Georgia Institute of Technology

Srinivas Aluru
Georgia Institute of Technology

ABSTRACT

Transmission, storage, and archival of high-throughput sequencing (HTS) short-read datasets pose significant challenges due to the large size of such datasets. Constant improvements to HTS technology, in the form of increasing throughput and decreasing cost, and its increasing adoption amplify the problem. General-purpose compression algorithms have been widely adopted for representing read datasets in a compact form. However, they are unable to fully leverage the domain-specific properties of read datasets. In response, researchers proposed special-purpose compression algorithms which improve upon the compression efficiency of general-purpose compression algorithms. In this paper, we present ParRefCom, a parallel *reference-based* algorithm for compressing HTS genomics short-read datasets. HTS instruments are typically used to generate paired-end reads as they hold significance for biological analysis. In contrast to existing special-purpose compression algorithms, ParRefCom treats *paired-end* reads as first-class citizens. Owing to this treatment of paired-end reads, our algorithm is able to significantly improve compression efficiency over the state-of-the-art. More specifically, for a benchmark human dataset, the size of the compressed output is 21% smaller than that produced by the current best algorithm. Further, ParRefCom is scalable and its compression and decompression speeds are better than those of *reference-free* methods.
Implementation : https://github.com/ParBLiSS/refcom

ACM Reference format:
Nagakishore Jammula and Srinivas Aluru. 2019. ParRefCom : Parallel Reference-based Compression of Paired-end Genomics Read Datasets. In *Proceedings of 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, Niagara Falls, NY, USA, September 7–10, 2019 (ACM-BCB '19),* 10 pages.
https://doi.org/10.1145/3307339.3342171

1 INTRODUCTION

Since the introduction of high-throughput sequencing (HTS) machines, the cost of sequencing has been declining and the throughput has been increasing exponentially [15]. For instance, using the recent NovaSeq line of instruments from Illumina, the current market leader, sequencing cost is expected to come down to $100 per human genome. Transmission, storage, and archival of HTS short-read datasets pose significant challenges owing to the large size of such datasets. Constant improvements to sequencing technology, in the form of increasing throughput and decreasing cost, and its growing adoption for a wide variety of applications amplify the problem. In response to this problem, researchers resorted to compression of read datasets.

General-purpose compression algorithms have been widely adopted for representing HTS read datasets in a compact form. Read datatsets have several unique properties that make it difficult for general-purpose compressors to fully exploit the redundancy present in these datasets. Domain-specific properties of read datasets include reduced size of alphabet, interleaved streams of data, fixed length for reads, occurrence of reads and their reverse-complements, paired representation of reads, scattered nature of redundancy, and availability of reference sequence. Researchers proposed special-purpose compression algorithms, that exploit one or more of these properties, to improve upon the compression efficiency of general-purpose compressors. Based on whether or not a reference sequence is made use of during compression, specialized compressors can be classified as reference-based or reference-free, respectively.

In this work, we leverage all of the above mentioned domain-specific properties of read datasets to develop ParRefCom, a parallel reference-based compressor for genomics read datasets. Read datasets generated using HTS instruments widely deployed today typically contain what are known as *paired-end* reads. A paired-end (PE) read is comprised of two separate but related reads, and the pairing information can serve to be crucial during biological analysis. Our specialized compressor allows the following lossless transformations of PE reads in the datasets : reordering of reads while keeping paired ends together and reordering of individual reads within a PE read. By exploiting these insights, ParRefCom is able to significantly improve upon the compression efficiency over state-of-the-art. More specifically, for a benchmark human dataset, the size of the compressed output is 21% smaller than that produced by SPRING [4], the current best algorithm.

At a high-level, our solution approach consists of the following steps: (1) Specialized alignment of PE reads to standard reference, (2) Classifying PE reads based on the number of ends aligned, and (3) Customizing compression strategies for reads in different categories. In this work, we develop fast and scalable parallel algorithms for accomplishing each of these tasks. We demonstrate that ParRefCom achieves superior compression efficiency compared to existing methods. Our compressor is asymmetric by design - decompression speed is much faster than compression speed. This asymmetricity in design goes well with the real world requirement of compressing a dataset once and decompressing (using) it many times.

Reference-based compression algorithms tend to achieve superior compression efficiency as they are able to leverage external knowledge in the form of reference sequence. On the other hand, reference-free compression algorithms tend to achieve superior

ACM-BCB '19, September 7–10, 2019, Niagara Falls, NY, USA

ACM ISBN 978-1-4503-6666-3/19/09…$15.00
https://doi.org/10.1145/3307339.3342171

compression speed as they avoid the computationally expensive step of base-to-base alignment. As ParRefCom makes use of a fast and scalable specialized alignment algorithm, it combines the best of both worlds and offers high compression efficiency and speed.

2 BACKGROUND

2.1 Sequencing and Representation

A DNA molecule is comprised of two strands. The forward strand of the molecule, defined in the $5'$ to $3'$ direction, is modeled as a sequence of characters from the alphabet set $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. Given such a sequence of l characters, $s = s_1, \ldots, s_l$, its reverse complementary strand *i.e.,* the sequence in the $3'$ to $5'$ direction, is $\bar{s} = c(s_l), \ldots, c(s_1)$, where $c(x), x \in \Sigma$ is the mapping function : $c(\mathtt{A}) \rightarrow \mathtt{T}, c(\mathtt{C}) \rightarrow \mathtt{G}, c(\mathtt{G}) \rightarrow \mathtt{C}$, and $c(\mathtt{T}) \rightarrow \mathtt{A}$. High-throughput sequencing (HTS) instruments are used to sequence a large number of randomly generated fragments from the genome. A few hundred bases are sequenced from these fragments and are commonly referred to as *reads*.

Most widely deployed HTS instruments from Illumina are capable of generating *paired-end reads*. A paired-end read consists of two reads which are sequenced from opposite ends of a DNA fragment, referred to as *insert*. Further, it is typical to sequence one of the reads from the forward strand and the other from the reverse complementary strand. Pairing information can serve to be crucial during biological analysis. HTS instruments, owing to their limitations, fail to accurately decipher bases sometimes. When inference is ambiguous or unsuccessful, an N character is recorded in the read. For this reason, the DNA alphabet set is expanded to include N for read datasets and $c(\mathtt{N}) \rightarrow \mathtt{N}$. Each base in the genome is typically spanned by multiple reads. This oversampling or redundancy is required to facilitate subsequent analysis of the reads. *Coverage* is defined as the average number of reads spanning a base in the genome. A k-length DNA sequence is termed a k-mer.

2.2 FASTQ File Format

Read datasets generated using HTS instruments are typically represented as FASTQ files [5]. A FASTQ file contains the following pieces of information for every single-end read : Read identifier, Bases constituting the read, Comment line, and Quality score corresponding to each base. Read identifiers are typically not made use of during analysis. Further, due to their structure, it is relatively straightforward to represent the identifiers compactly. Prior works explored such representations. Comments are either empty or exactly identical to identifier lines. Significant efforts were devoted to develop standalone compressors for quality scores. A recent work explored use of a single bit to capture a quality score value [17]. Further, the work demonstrated that such lossy compression of metadata does not have any noticeable effect on biological analysis. Due to these reasons, in this work, we focus on compactly representing the read data in FASTQ files in a lossless manner.

2.3 General-purpose Vs. Special-purpose Compression

General-purpose compression algorithms have been widely adopted for representing HTS read datasets in a compact form, with ZIP family of algorithms and their blocked variants being the prominent examples [18]. Domain-specific properties of read datasets make it difficult for general-purpose compressors to fully exploit the redundancy present in these datasets. Alphabet size of characters occurring in reads is small, typically 5. FASTQ files contain several interleaved streams of data as described in Section 2.2. Reads generated by HTS instruments mostly have fixed length. Reads and/or their reverse-complements can represent segments of DNA in datasets. Reads are represented as a related pair in case of paired-end read datasets. Redundancy in sequencing, necessary to facilitate subsequent analysis of reads, is scattered across the dataset. Differences between genomes of organisms belonging to a species are typically very small. Therefore, genome of an organism of a species can be made use of to compactly represent the read dataset of another organism belonging to the same species. Researchers proposed special-purpose compression algorithms, that exploit one or more of the above properties, to improve upon the compression efficiency of general-purpose compressors. Based on whether or not a reference sequence is made use of during compression, specialized compressors can be classified as reference-based or reference-free, respectively. In the following section, we furnish a survey of special-purpose compression algorithms proposed for HTS short read datasets.

3 RELATED WORK

A number of special-purpose compression tools have been proposed over the years for compression of FASTQ datasets, both in reference-free and reference-based categories. These include DSRC [16], Fqzcomp [2], Fastqz [2], FQC [6], SCALCE [7], LW-FQZip [8], Quip [9], Leon [1], k-Path [10], and Mince [14]. A recent review article evaluated tools with publicly available implementations using a set of benchmark datasets [13]. The review article also provides short descriptions of the tools mentioned previously. Three special-purpose compression tools, which demonstrated better compression efficiency, have been proposed since the review article was published - FASTORE [17], minicom [12], and SPRING [4].

FASTORE clusters reads, i.e. distributes them into bins, such that reads from neighboring positions are likely to belong to the same cluster. Within each bin, reads are matched to generate a *reads similarity graph*. After this, reads go through an optional step of redistribution and matching. Reads are then assembled into contigs using the final similarity graph. Based on the outcome of the matching, reads are encoded with respect to contigs or other reads. *minicom* also takes an approach similar to that of FASTORE. It additionally attempts to merge contigs to build longer contigs. SPRING tool comprises of the following steps : Reordering reads according to their position in the genome using *hashed substring indices*, Encoding reordered reads to remove redundancy between consecutive reads, and Compressing encoded reads using general-purpose compression tools. In the following section, we provide a high-level overview of our solution approach.

4 OVERVIEW OF SOLUTION APPROACH

In this work, we leverage all of the previously mentioned domain-specific properties of read datasets to develop a reference-based compressor for genomics read datasets. The ordering of paired-end

(PE) reads in a dataset and the ordering of two reads of a PE read do not carry any special significance. Therefore, our specialized compressor allows the following lossless transformations of PE reads in datasets : reordering of reads while keeping paired ends together and reordering of individual reads within a PE read. By utilizing these insights, ParRefCom is able to significantly improve upon the compression efficiency over state-of-the-art.

We developed a solution approach that leverages all of the properties described in Section 2.3 and the insights mentioned above. A high-level description of the overall approach is as follows. First, we perform a specialized alignment of PE reads in the dataset using standard reference for the species. Next, we classify the PE reads based on the outcome of the alignment. The categories are : Two-aligned PE read (when both ends of the PE read align), One-aligned PE read (when one of the ends of the PE read aligns but the other does not), and Non-aligned read (when both ends of the PE read do not align). Finally, we develop custom compression strategies for each of these categories. Through evaluation using an assortment of read datasets, we demonstrate that PE reads in the first category (Two-aligned) significantly outnumber those in the remaining two categories.

In the output of the specialized compressor, we capture One-aligned and Non-aligned PE reads as they are. For Two-aligned PE reads, we capture the following pieces of information in place of the reads themselves : (1) Starting location with respect to the reference sequence, (2) Number of differences with respect to the reference sequence, (3) Positions of differences within a read, (4) Bases corresponding to differences within a read and (5) Locations of other ends for one of the ends. Collectively, these pieces of information are sufficient to reconstruct the original PE reads as depicted in Figure 1.

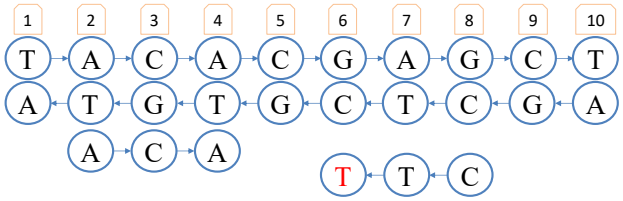

Figure 1: A fixed-length single-end read can be recovered by knowing the starting location where the read aligns to the reference, the number of alignment differences, the positions of these differences, and the differing bases. Further, a paired-end read can be recovered from two single-end reads by knowing the location of the other end for one of the ends

In the following sections, we develop fast and scalable parallel algorithms for accomplishing tasks that make up our solution approach. By utilizing these algorithms, we demonstrate that ParRefCom achieves superior compression efficiency compared to existing methods. Our compressor is asymmetric by design - decompression speed is much faster than compression speed. This is because the decompression step does not involve the computationally expensive specialized alignment phase. This asymmetricity in design goes well with the real world requirement of compressing a dataset once and decompressing (using) it many times. We describe our experimental methodology next.

5 METHODOLOGY

Table 1 lists key properties of datasets used for evaluating this work. The datasets correspond to organisms from three different species with varying genome lengths. These diverse datasets are publicly available and were previously used to benchmark many special-purpose compression tools. Column 2 represents the number of fixed-length paired-end reads in each dataset. The length of these reads is captured in Column 3 and sizes of the original datasets in Column 4. Among the datasets used for benchmarking various tools in [13], H. sapiens and T. cacao correspond to big genomics datasets. We included the former in our evaluation. The latter was excluded as it is very old and the two ends of paired-end reads have different lengths.

Table 1: Datasets used for experimental evaluation

Organism	Spots/Inserts (Thousands)	Read length (Bases)	Original Size
C. elegans	33,809	2 × 100	6.8 GB
G. gallus	173,698	2 × 100	34.7 GB
H. sapiens (H1)	24,476	2 × 100	4.9 GB
H. sapiens (H2)	207,680	2 × 101	42 GB
H. sapiens (H3)	270,765	2 × 146	79 GB

Accession numbers for C. elegans, H. sapiens (H1), and H. sapiens (H2) are SRR065390, SRR062634, and ERR174310 respectively. Accession numbers for G. gallus dataset are - SRR197985, SRR197986, SRR105788, SRR105789, SRR105792, and SRR105794. H. sapiens (H3) dataset was generated using Illumina NovaSeq and is available from Illumina BaseSpace as NA12878-Rep-1_S1_L001. This dataset contains variable length reads with length up to 151. We trimmed the reads down to 146 bases to make them fixed-length while retaining the maximum number of reads. The approximate sizes of the standard reference genome for C. elegans, G. gallus, and H. sapiens are 100, 1125, and 3300 mega base-pairs respectively. Reference sequences used for respective organisms are - GCF_000002985.6, GCF_000002315.4, and GCF_000001405.38. We use bsc (http://libbsc.com) to compress several streams of information generated in Sections 7 and 8.

We ran our experiments on a machine with two 14-core Intel Xeon processors, for a total of 28 cores. The 28 cores in a node share 256 GB of main memory. Further, we used POSIX threads for shared-memory parallel programming. For reporting performance results, each experiment was repeated three times. The wall-times were collected for each run, and the minimum times amongst the repeats of an experiment are reported as they closely represent the capabilities of the system.

6 SPECIALIZED ALIGNMENT

6.1 Motivation

The goal of reference-based special-purpose compression algorithms is to represent the read dataset compactly by capturing

differences in reads with respect to reference genome. Our specialized compressor, ParRefCom, also employs a reference-based strategy. Aligning reads in a dataset, generated from sequencing a target genome, to standard reference is a fundamental bioinformatics problem. The objective of read alignment is to glean biological insights by computing and analyzing variations between target genome and reference genome.

Some attributes of classical alignment are detrimental to compression efficiency when the objective is to represent a read dataset compactly. When there are multiple equally good alignments for a read, classical alignment tools may report more than one alignment for such reads. For compression, one best alignment suffices. Generating alternative alignments for reads also involves a computational overhead. While aligning paired-end (PE) reads, classical alignment tools perform analysis to estimate the insert size and discard *abnormal* alignments [3]. For example, when aligning a PE read, the location of the read whose reverse-complement aligns to the reference genome is expected to be after that of the read that aligns as is. If this is not the case, the alignment may be discarded. As we are not concerned about biological significance during compression, the compression efficiency would improve if such an alignment were to be accepted. Finally, clipping of reads performed by classical alignment tools may impede faithful recovery of such reads.

In this section, we propose a specialized alignment algorithm (SAA) that addresses the drawbacks of the classical alignment algorithms when the objective of the alignment is to represent a read dataset compactly. The goal of our SAA is as follows. For a PE read, we want to generate an alignment that minimizes the following expression : $D_1 + D_2 + S_{12}$; D_1 : Differences between first read and reference genome, D_2 : Differences between second read and reference genome, and S_{12} : Separation between alignment locations of two reads. We propose an SAA utilizing kmer-index that optimizes the stated objective function. Next, we describe the kmer-index data structure and provide a parallel algorithm for constructing it.

6.2 Index Data Structure

Our kmer-index data structure comprises of two levels, each an array, as depicted in Figure 2. Level 2 (L2) array consists of locations of all kmers in the reference genome. The entries are sorted by kmer as primary key, and for a kmer by location as secondary key. The size of level 1 (L1) array is $4^k + 1$, k : kmer size. Each entry in this array, excluding the last one, corresponds to a kmer and captures the starting position of the corresponding kmer in L2 array. If a kmer does not occur in the reference genome, it does not have any entries corresponding to it in L2 array. For such kmers, the corresponding L1 array entry contains the same value as that of the next kmer. To summarize, two consecutive entries in L1 array help determine the number of occurrences of a kmer in the genome. Values recorded in L2 array in the range defined by the two entries provide the corresponding kmer locations.

Using a two-level index data structure has several advantages. The number of occurrences of a kmer in the reference genome can be determined by accessing two consecutive entries in L1 array. For kmers not occurring in the reference, the computed frequency is 0. This quick determination can aid in reducing the alignment time. Further, the locations where the kmer occurs in the genome

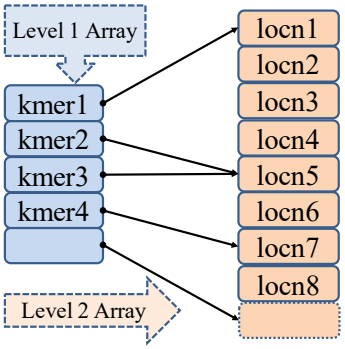

Figure 2: An illustration of two-level kmer-index data structure. kmer1, kmer3, and kmer4 occur four, two, and two times respectively in the reference genome. kmer2 does not occur in the reference genome

are present in a contiguous segment of L2 array. Scanning these locations therefore benefits from spatial locality of cache and/or memory accesses. Memory consumption of the two-level index data structure is $(4^k + 1) \times 2k + (|\mathcal{G}| - k + 1) \times \lceil \log_2 |\mathcal{G}| \rceil$ bits, where $|\mathcal{G}|$ represents the length of the genome. For large genomes and small values of k (< 15), as used in this work, second term dominates. The overhead due to L1 array is insignificant. Therefore, the two-level index data structure enables fast kmer lookups while incurring a small memory overhead. Next, we present a parallel algorithm for constructing the two-level index data structure.

6.2.1 Parallel Index Construction. Algorithm 1 demonstrates the parallel construction of the two-level index data structure. Initially, reference genome is block decomposed among the available threads. Every thread generates the list of $(kmer, locn)$ tuples for the block owned by the thread. Next, the list across all threads is sorted in parallel. *locn*s from the resulting sorted list make up L2 array. L1 array is obtained by performing a parallel prefix on the sorted list of tuples.

6.3 Alignment Algorithm

The objective of our specialized alignment algorithm (SAA) is as follows. For a paired-end (PE) read, we want to generate an alignment that minimizes the following expression : $D_1 + D_2 + S_{12}$; D_1 : Differences between first read and reference genome, D_2 : Differences between second read and reference genome, and S_{12} : Separation between alignment locations of two reads. In this section, we describe the design of such an SAA and its parallel implementation. In case of a PE read, we assume that one of the ends is sequenced from the forward strand and the other from the reverse-complementary strand. This assumption is representative of the most common form of paired-end sequencing, called forward-reverse sequencing.

Let e denote the number of differences we want to tolerate between a read and the reference genome. According to pigeon-hole principle, if the read is decomposed into $(e + 1)$ non-overlapping subsequences, then there is at least one subsequence of the read that matches exactly with an identical length subsequence from the reference genome. This property can be extended as follows. If the

Algorithm 1: Parallel construction of kmer-index

Input: $\mathcal{G}$, reference genome. k, kmer size. t, thread count
Output: Two-level kmer-index: L1 and L2 arrays
// Size of L1 is $4^k + 1$
// Size of L2 is $|\mathcal{G}| - k + 1$. $|\mathcal{G}| \leftarrow$ genome size

parallel *j = thread's id* **do**
  $S_G \leftarrow$ Size of the reference genome $\mathcal{G}$
  $L_G \leftarrow \left\lceil \frac{S_G}{t} \right\rceil$
  $lt \leftarrow j \times L_G$
  $rt \leftarrow (j+1) \times L_G$ - 1
  if $(S_G - k) < rt$ **then**
    $rt \leftarrow (S_G - k)$
  end
  Initialize $\mathcal{T}$ to an empty list of tuples
  for $i \leftarrow lt$ **to** rt **do**
    Append $(kmer, locn)$ to $\mathcal{T}$
  end
  Parallel sort $\mathcal{T}$ using *kmer* as primary key and *locn* as secondary key
  for $i \leftarrow lt$ **to** rt **do**
    L2[i] $\leftarrow \mathcal{T}[i].locn$
  end
  Use parallel prefix to populate L1 with start position of every kmer in $\mathcal{T}$
end

read is decomposed into $(e+2)$ non-overlapping subsequences, then there are at least two subsequences of the read that match exactly with identical length subsequences from the reference genome. The extended pigeon-hole principle has the ability to filter out spurious matches more effectively. Note that the extension does not have any impact on the alignment output but only serves to potentially reduce the computational cost. Therefore, we adopt the extended pigeon-hole principle in our SAA.

A high-level overview of our SAA is as follows. First, we decompose a read into non-overlapping kmers. We look up each kmer in L1 array to determine its frequency of occurrence. We select a subset of $(e + 2)$ kmers that occur with the lowest frequency. For these $(e + 2)$ kmers, we look up their corresponding locations in L2 array. Next, we obtain a subset of locations which correspond to at least two kmers. We perform a banded-alignment of the read at the locations in the subset using a vectorized Meyer's bit-vector algorithm. If the read aligns to the reference sequence with $\leq e$ differences, we record the start position of the alignment and the differences.

For a PE read, we perform the above steps for each individual read and its reverse-complement. From this point on, for simplicity, we refer to a read that aligns as is to the reference sequence as forward read and a read whose reverse-complement aligns to the reference sequence as reverse read. We then select the best alignment as one that minimizes the sum of (1) Differences between forward read and reference genome, (2) Differences between reverse read and reference genome, and (3) Separation between alignment start locations of the two reads. Further, we classify PE reads based on the outcome of the alignment as : Two-aligned (when both ends of the PE read align), One-aligned (when one of the ends of the PE read aligns but the other does not), and Non-aligned (when both ends of the PE read do not align). We present our strategies for handling reads in each of the three categories in subsequent sections. The parallel implementation of our SAA is described in Algorithm 2.

Algorithm 2: Parallel specialized alignment algorithm

Input: $\mathcal{R}$, paired-end read dataset. k, kmer size. t, thread count
Input: $\mathcal{B}$, read block size. e, number of differences
Input: kmer-index, L1 and L2 arrays
Output: LF_{Two}: List of forward two-aligned reads
Output: LR_{Two}: List of reverse two-aligned reads
Output: L_{One} : List of paired-end one-aligned reads
Output: L_{Non} : List of paired-end non-aligned reads
// Reads in LF_{Two} and LR_{Two} correspond one to one

parallel *j = thread's id* **do**
  while *reads in* $\mathcal{R}$ **do**
    Parse $\mathcal{B}$ reads from $\mathcal{R}$
    for *each r in* $\mathcal{B}$ **do**
      $r1 \leftarrow$ first read in r
      $r2 \leftarrow$ second read in r
      Align $r1$ and $r2$ as described in Section 6.3
      if $r1$ *and* $r2$ *align* **then**
        $rf \leftarrow$ forward read. $rr \leftarrow$ reverse read
        Append $(rf, locnf, id)$ to LF_{Two}
        Append $(rr, locnr, id)$ to LR_{Two}
      else if $r1$ *or* $r2$ *aligns* **then**
        if $r1$ *aligns as forward read* **then**
          $rf \leftarrow r1$. $rr \leftarrow$ reverse of $r2$
          Append $(rf, rr, locn1)$ to L_{One}
        else if $r1$ *aligns as reverse read* **then**
          $rf \leftarrow r2$. $rr \leftarrow$ reverse of $r1$
          Append $(rf, rr, locn1)$ to L_{One}
        else if $r2$ *aligns as forward read* **then**
          $rf \leftarrow r2$. $rr \leftarrow$ reverse of $r1$
          Append $(rf, rr, locn2)$ to L_{One}
        else
          $rf \leftarrow r1$. $rr \leftarrow$ reverse of $r2$
          Append $(rf, rr, locn2)$ to L_{One}
        end
      else
        Determine rf, rr, and *locn* as described in Section 8
        Append $(rf, rr, locn)$ to L_{Non}
      end
    end
  end
end

SAA lends itself very well to parallelization. PE read dataset $\mathcal{R}$ is decomposed into virtual blocks, each of size $\mathcal{B}$ reads. Each thread

parses $\mathcal{B}$ reads, performs alignment for them, and generates the corresponding output. The output blocks are appended to appropriate lists as described in Algorithm 2. Threads only need to synchronize to determine the blocks of reads to work on. Each thread seeks ownership of a new block of reads once it is done working on the block it currently owns. The parameter $\mathcal{B}$ can be tuned appropriately based on the number of threads to ensure none of the threads starves. Note that a straightforward block decomposition of reads among threads can lead to a load imbalance among threads. This is because the computational effort necessary to align reads varies from one read to another.

6.4 Results and Analysis

In this section, we present results obtained using our specialized alignment algorithm (SAA). Before proceeding to discuss the results, we furnish the values used for parameters and the rationale behind selecting the specific values. For the number of hardware threads typical among shared-memory machines, a value of 4000 for $\mathcal{B}$, read block size, ensures that threads spend almost all their time performing useful work, namely read alignment. The value of $\mathcal{B}$ is independent of the dataset.

A value of 14 for k allows us to tolerate up to 5 differences per 100 bases ($\lfloor \frac{100}{(5+2)} \rfloor$) between a read and the corresponding subsequence of the reference genome. Note that SAA makes use of extended pigeon-hole principle. We choose a small value of 3 for e initially, to reduce the computational overhead. However, our implementation of the banded and vectorized Meyer's bit-vector algorithm allows a band size of up to 7. We utilize the full potential of the alignment algorithm to allow up to 7 differences. So, the final value of e supported by SAA is 7. Finally, for every read, we explore up to 2000 potential locations to select the best alignment. Note that the ability to align a read, on average, tapers off as the number of explored locations increases. The chosen value helps achieve a good trade off between the number of reads aligned and the computational cost incurred.

Table 2 shows the percentage of paired-end (PE) reads falling into each of the three categories : two-aligned, one-aligned, and non-aligned, for all datasets using SAA. It can be observed that the percentage of two-aligned reads is the highest for all datasets, with the value for human dataset reaching 90%. PE reads in this category offer the most potential for compression. We present our strategy for compressing reads in two-aligned category in Section 7. It should be noted that C. elegans and G. gallus datasets were generated using older generation of sequencing instruments. Therefore, these datasets contain more sequencing errors, which are partly responsible for the lower percentage of PE reads in two-aligned category compared to human datasets. As we mentioned previously, these results correspond to the case where we tolerate up to 7 differences per single-end read. Our strategy for compressing the reads falling into one-aligned and non-aligned categories is described in Section 8.

In Section 6.1, we described the rationale behind designing a specialized alignment algorithm to complement classical alignment algorithms when the objective is to represent the read dataset in a compact manner. Table 3 shows the percentage of PE reads falling into various categories for all datasets using BWA [11], a flagship classical alignment tool. It can be observed that the output of BWA comprises of two additional categories : clipped and multi-aligned. For PE reads in the clipped category, we do not have complete alignment for at least one of the ends. In case of multi-aligned category, we have multiple alignments for at least one of the ends. Even thought the percentage of two-aligned PE reads is higher for some datasets, the reads in clipped and multi-aligned categories pose challenges for compression. BWA was run with default parameters and tolerates more differences per single-end read than SAA, which is mostly responsible for the higher percentage of two-aligned PE reads.

Table 2: Percentage of paired-end reads aligned using specialized alignment algorithm

Dataset	Two-aligned	One-aligned	Non-aligned
C. elegans	85.12	8.17	6.71
G. gallus	84.14	5.87	9.99
H. sapiens (H1)	88.57	8.21	3.22
H. sapiens (H2)	90.04	7.94	2.02
H. sapiens (H3)	88.46	8.84	2.70

Table 3: Percentage of paired-end reads aligned using BWA

Dataset	2-align	1-align	0-align	Clipped	Multi-aligned
CE	81.63	0.65	3.85	12.71	1.16
GG	80.07	0.24	0.45	18.71	0.53
H1	89.88	0.23	0.16	9.27	0.46
H2	93.94	0.23	0.42	5.06	0.35
H3	94.26	0.09	0.31	4.58	0.76

Table 4 shows the time taken by BWA and SAA for alignment using identical number of threads, 16 in this case. It can be noticed that SAA is nearly an order of magnitude faster than BWA. This demonstrates the superior computational capability of SAA compared to classical alignment tools for the purpose of representing read datasets compactly. Index construction time is not included for both BWA and SAA. The value in case of SAA for C. elegans, G. gallus, and H. sapiens is 3, 51, and 103 seconds respectively. In the following section, we describe our algorithm and its parallel implementation for handling PE reads in the two-aligned category.

Table 4: Alignment time in seconds using 16 threads for BWA and specialized alignment algorithm

Dataset	BWA time (s)	SAA time (s)	SAA speedup
C. elegans	413	46	8.98
G. gallus	5265	433	12.16
H. sapiens (H1)	922	101	9.13
H. sapiens (H2)	8140	852	9.55
H. sapiens (H3)	16054	1770	9.07

7 HANDLING TWO-ALIGNED PAIRED-END READS

7.1 Overall Approach

For two-aligned paired-end (PE) reads, we capture the following pieces of information in lieu of the reads themselves : (1) Alignment start location with respect to the reference sequence, (2) Number of differences with respect to the reference sequence, (3) Positions of differences within a read, (4) Bases corresponding to differences within a read and (5) Locations of other ends for one of the ends. Collectively, these pieces of information are sufficient to reconstruct two-aligned PE reads. In the following subsections, we provide a detailed description for generating each of these pieces of information.

Algorithm 3 demonstrates parallel generation of data streams to be recorded for two-aligned PE reads. Lists of forward and reverse two-aligned reads generated by Algorithm 2 are independently sorted in parallel based on the alignment locations of the reads. The sorted lists are block decomposed among available threads and each thread is responsible for generating data streams for reads in the block owned by it. Data stream (5) is an exception and we describe the process for generating it in Section 7.6. Note that data streams (1)-(4) are generated for both forward and reverse reads and data stream (5) only for forward reads. This is because it is sufficient to record pairing information only for one of the ends, forward read in our case, of a PE read. The generated data streams are compressed in parallel using bsc, a general-purpose compressor.

7.2 Generating List of Starting Locations

The first piece of information that needs to be recorded for every read, forward and reverse, is the location in the reference genome where the read starts aligning with the genome. There are two possible options for recording the alignment start location information : (1) Combine forward and reverse reads into a single list or (2) Maintain them in separate lists. In the former case, in addition to storing the start location, a bit is necessary to indicate whether the read is a forward read or a reverse read. Further, there can be implication for how the pairing information is maintained. Owing to these reasons, it is beneficial to choose the second option. For a read, we store its relative start location, location delta from previous read, instead of the absolute start location. This offers additional space savings.

For all datasets, and for both forward and reverse lists, the frequency of occurrence of relative start location values falls off sharply as the values increase. Based on this observation, we use the following scheme to encode location deltas. We use one byte to encode the location delta if the value is < 253. We use the remaining three values that can be represented using a byte - $253, 254$, and 255 - to indicate that $2, 3$, and 4 bytes are necessary to capture the location delta value respectively. The value is stored using the corresponding number of bytes.

7.3 Generating List of Number of Differences

The next piece of information that needs to be recorded for every read is the number of bases different in the alignment between the read and the reference genome. We use one byte to capture each difference count.

Algorithm 3: Parallel generation of data streams for two-aligned paired-end reads

Input: LF_{Two}: List of forward two-aligned reads
Input: LR_{Two}: List of reverse two-aligned reads
Output: LF_{SL} and LR_{SL}: List of starting locations
Output: LF_{ND} and LR_{ND}: List of number of differences
Output: LF_{PD} and LR_{PD}: List of positions of differences
Output: LF_{BD} and LR_{BD}: List of bases corresponding to differences
Output: L_{PE}: List of locations of other ends
// LF_{Two} and LR_{Two} have the same size
// They are lists of tuples of type $(read, locn, id)$
1 **parallel** j = *thread's id* **do**
2 $S_G \leftarrow$ Size of LF_{Two}. $L_G \leftarrow \left\lceil \frac{S_G}{t} \right\rceil$
3 $lt \leftarrow j \times L_G$. $rt \leftarrow (j+1) \times L_G$ - 1
4 **if** $(S_G - 1) < rt$ **then**
5 $rt \leftarrow (S_G - 1)$
6 **end**
7 Parallel sort LF_{Two} using *locn* as key. Parallel sort LR_{Two} using *locn* as key
8 Initialize $\mathcal{T}F$ to an empty list of tuples. Initialize $\mathcal{T}R$ to an empty list of tuples
9 **for** $i \leftarrow lt$ **to** rt **do**
10 Append $(idf, locnf, posnf)$ to $\mathcal{T}F$
11 Append $(idr, locnr)$ to $\mathcal{T}R$
12 Append starting location of $LF_{Two}[i]$ to LF_{SL}
13 Append starting location of $LR_{Two}[i]$ to LR_{SL}
14 Append number of differences of $LF_{Two}[i]$ to LF_{ND}
15 Append number of differences of $LR_{Two}[i]$ to LR_{ND}
16 Append positions of differences of $LF_{Two}[i]$ to LF_{PD}
17 Append positions of differences of $LR_{Two}[i]$ to LR_{PD}
18 Append differing bases of $LF_{Two}[i]$ to LF_{BD}
19 Append differing bases of $LR_{Two}[i]$ to LR_{BD}
20 **end**
21 Parallel sort $\mathcal{T}F$ using *id* as key. Parallel sort $\mathcal{T}R$ using *id* as key
22 Initialize $\mathcal{T}P$ to an empty list of tuples
23 **for** $i \leftarrow lt$ **to** rt **do**
24 Append $(locnf, locnr, posnf)$ to $\mathcal{T}P$
25 **end**
26 Parallel sort $\mathcal{T}P$ using *locnr* as key
27 **for** $i \leftarrow lt$ **to** rt **do**
28 Append $(locnr - loncnf, posnf)$ to L_{PE}
29 **end**
30 **end**

7.4 Generating List of Positions of Differences

In addition to storing the counts of differences, we need to record the positions of differing bases for reads which have one or more

differences. The number of bits necessary to store each difference is $log_2 L$, where L denotes the length of the read. This cost can be brought down if we store relative difference positions. We use one byte to encode each relative difference position.

7.5 Generating List of Bases Corresponding to Differences

In order to recreate individual reads exactly during decompression, we need to record the bases corresponding to the differences along with the counts and the positions of the differences. In our experiments, we observed that the count of substitutions significantly outnumber the counts of insertions and deletions. Further, the counts of insertions and deletions are approximately the same.

Taking these characteristics into account, we developed the following encoding scheme to capture the base differences. We use 2 bits for capturing a substitution. Note that `A`, `C`, `G`, `T` can be substituted with one of the other three bases. This leaves us with one unused value, 11, which can be used to capture additional scenarios using 2 more bits. The four values made available through the additional 2 bits are used to capture : substitution to an `N`, deletion, insertion of an `N`, and insertion of a regular base. When the insertion corresponds to a regular base, we use 2 additional bits to capture the inserted base.

7.6 Generating List of Locations of Other Ends

Collectively, data streams (1) - (4) described in preceding subsections contain sufficient information to recover forward and reverse reads independently. We now describe the information we capture so that reverse reads can be paired with their corresponding forward reads. Among the approaches available to capture pairing information, the one that incurs the least cost is the following. For every reverse read, we capture the relative location of its forward read. There can be more than one forward read that aligns starting at a given location. To account for such cases, we capture which of these forward reads pairs with the reverse read under consideration.

We use one byte to encode each value of the second type. The former values follow a normal distribution. We map the distribution mean to zero and compute the corresponding folded-normal distribution. We use the scheme described in Section 7.2 to encode the resulting values. This concludes our discussion on capturing the data streams necessary for encoding and decoding two-aligned PE reads. Next, we discuss our approach for handling one-aligned and non-aligned PE reads.

8 HANDLING ONE- AND NON- ALIGNED PAIRED-END READS

In Section 6.4, we showed that there is a small fraction of paired-end (PE) reads for which one or both ends do not align. We classified them under one-aligned and non-aligned categories, respectively. Even though we do not have the desirable outcome for these PE reads, our specialized alignment algorithm (SAA) generates sufficient information to orient and order them. In order to orient PE reads, we need to know which read to capture as forward read and which one to capture as reverse read. Relative placement of PE reads with respect to one another helps in determining a good ordering to assist in better compression. In case of one-aligned PE reads, we have an alignment for one of the ends of the PE read. Based on whether the end is aligned as a forward read or as a reverse read, we have the necessary information to determine the orientation. Refer to Algorithm 2. Further, location of the aligned end assists in ordering the reads relative to one another.

The determination of orientation and ordering of non-aligned PE reads works as follows. Note that we don't have an alignment for any of the ends in case of a non-aligned read. When we attempt to align a PE read using SAA, we generate potential candidate locations where the read may align. When such candidate locations are available for a non-aligned PE read, we pick the best among available locations and use it to orient and order reads. We label such PE reads under non-aligned-x category. Even when potential candidate locations are not available, we can have one or more successful kmer lookups. When information from looking up kmers is available, we pick the best among such lookups and use it to orient and order reads. We label such PE reads under non-aligned-y category. Finally, when we have no information available to orient and order PE reads, we label such reads under non-aligned-z category.

The orientation for PE reads in non-aligned-z category is determined based on the order in which reads appear in the dataset, first read as forward read and second read as reverse read. In terms of ordering PE reads in this category, we place them after PE reads belonging to the remaining categories. In Section 10, we will show that the percentage of reads in non-aligned-z category is zero or close to it for all datasets. Therefore, even when both ends of a PE read do not align, we have sufficient evidence to orient and order reads.

For one- and non-aligned PE reads, orientation is determined while attempting to align the reads using SAA. Further, they are ordered by performing a parallel sort. We use one byte per base encoding to capture one- and non- aligned PE reads. Oriented and ordered one- and non- aligned PE reads are compressed in parallel using bsc, a general purpose compressor. We considered alternative encodings, including using two bits for four regular bases and recording Ns separately, but one byte per base encoding yielded the best compression efficiency. It must be noted that classical alignment algorithms do not provide any information for orienting and ordering non-aligned reads.

9 DECOMPRESSION ALGORITHM

Our special-purpose compressor, ParRefCom, is asymmetric by design - decompression is much faster than compression. This is because the decompression step does not involve the expensive specialized alignment phase. This asymmetric design of ParRefCom goes well with the real world requirement of compressing a dataset once and decompressing (using) it many times. Our parallel algorithm for recovering the read dataset using the compressed representations described in Sections 7 and 8 works as described below. General-purpose compressor bsc is first used to perform parallel decompression. Recovery of one- and non- aligned paired-end (PE) reads is complete after this step. Note that reverse-complement of second end in every PE read is computed to undo the reverse-complement operation performed by the specialized alignment algorithm.

9.1 Recovering Two-aligned Reads

Our parallel algorithm for recovering two-aligned PE reads is described in Algorithm 4. We first use the reference genome and the list of starting locations to recover difference-free reads. Then, we use lists of counts of differences, positions of differences, and bases corresponding to differences in order to update bases that are different between reads and the reference genome. These steps are performed independently for lists corresponding to forward and reverse reads. Finally, we perform a parallel sort of reverse reads using information contained in data stream (5). This step accomplishes the task of pairing reverse reads with forward reads appropriately. The reference sequence used during decompression is the same as that used during compression.

Algorithm 4: Parallel algorithm for recovering two-aligned paired-end reads

Input: $\mathcal{G}$, reference genome. t, thread count
Input: C, number of two-aligned paired-end reads
Input: LF_{SL} and LR_{SL}: List of starting locations
Input: LF_{ND} and LR_{ND}: List of number of differences
Input: LF_{PD} and LR_{PD}: List of positions of differences
Input: LF_{BD} and LR_{BD}: List of bases corresponding to differences
Input: L_{PE}: List of locations of other ends
Output: LF_{Two}: List of forward two-aligned reads
Output: LR_{Two}: List of reverse two-aligned reads
// Reads in LF_{Two} and LR_{Two} correspond one to one
parallel *j = thread's id* **do**
  $S_G \leftarrow$ Size of LF_{Two}
  $L_G \leftarrow \left\lceil \frac{S_G}{t} \right\rceil$
  $lt \leftarrow j \times L_G$
  $rt \leftarrow (j+1) \times L_G$ - 1
  if $(S_G - 1) < rt$ **then**
    $rt \leftarrow (S_G - 1)$
  end
  for $i \leftarrow lt$ **to** rt **do**
    Populate $LF_{Two}[i]$ using decoded $LF_{SL}[i]$ and $\mathcal{G}$
    Populate $LR_{Two}[i]$ using decoded $LR_{SL}[i]$ and $\mathcal{G}$
    Identify differing bases in $LF_{Two}[i]$ using $LF_{ND}[i]$, and corresponding number of LF_{PD} entries
    Identify differing bases in $LR_{Two}[i]$ using $LR_{ND}[i]$, and corresponding number of LR_{PD} entries
    Update differing bases in $LF_{Two}[i]$ using $LF_{PD}[i]$, and corresponding number of decoded LF_{BD} entries
    Update differing bases in $LR_{Two}[i]$ using $LR_{PD}[i]$, and corresponding number of decoded LR_{BD} entries
    Compute reverse-complement of $LR_{Two}[i]$ read
  end
  Sort LR_{Two} in parallel using L_{PE} values as keys
  // $LF_{Two}[i]$ and $LR_{Two}[i]$ correspond to two ends of a paired-end read
end

10 RESULTS AND ANALYSIS

In this section, we present and analyze results corresponding to compression and decompression of all datasets. We use minicom and SPRING as baseline to compare our results. These are the most recent tools and demonstrated superior compression efficiency among the tools mentioned in Section 3. Table 5 shows the sizes of the compressed datasets, in mega bytes, produced by minicom, SPRING and ParRefCom. For comparison, the sizes of the original datasets are furnished in Table 1. It can be observed that ParRefCom performs better than SPRING for all datasets but C. elegans. Excluding the C. elegans dataset, the size of the compressed output produced by ParRefCom is at least 21% smaller compared to the next best. For human dataset H1, it is smaller by as much as 77%. These results demonstrate the superior compression efficiency of ParRefCom, our special-purpose compressor.

Table 5: Compressed file sizes (in MB) using minicom, SPRING, and ParRefCom algorithms

Dataset	minicom	SPRING	ParRefCom	Improvement
CE	249	**227**	267	-17.62%
GG	1,739	1,512	**1,175**	22.29%
H1	821	825	**192**	76.61%
H2	2,522	1,666	**1,324**	20.53%
H3	2,888	2,022	**1,459**	27.84%

Tables 6 and 7 depict the time taken by minicom, SPRING, and ParRefCom tools for compression and decompression respectively. The results are provided for all datasets and correspond to the case when 16 threads are used. For compression, ParRefCom is at least 1.9 times faster across all datasets. In case of decompression, ParRefCom is at least 1.4 times faster across all datasets. Note that a dataset typically needs to be compressed only once but needs to be decompressed multiple times. Therefore, decompression performance carries more significance than compression performance.

Table 6: Compression time in seconds using 16 threads for minicom, SPRING, and ParRefCom algorithms

Dataset	minicom	SPRING	ParRefCom	Speedup
CE	563	490	75	6.53
GG	12,680	2,374	634	3.74
H1	1,667	540	249	2.17
H2	16,232	2,502	1,117	2.24
H3	53,780	4,042	2,141	1.89

Table 8 shows the time taken by specialized alignment, compression, and decompression for human dataset H2 as the number of threads is varied from 2 to 16. Our algorithms demonstrate good scalability across the board. Most of the discrepancy in scalability is due to the sequential nature of IO, bsc, and interference from concurrently running jobs while accessing the file system. ParRefCom compression and decompression times reported in Tables 6, 7, and 8 are end-to-end times. Currently, ParRefCom performs all the

Table 7: Decompression time in seconds using 16 threads for minicom, SPRING, and ParRefCom algorithms

Dataset	minicom	SPRING	ParRefCom	Speedup
CE	31	33	19	1.63
GG	149	198	104	1.43
H1	51	56	21	2.43
H2	190	226	114	1.67
H3	273	326	187	1.46

processing in-memory. The memory consumption can be reduced by writing intermediate data to disk and using external-memory algorithms.

Table 8: Specialized alignment, compression, and decompression times in seconds for H. sapiens (H2) dataset using ParRefCom algorithm

No. of threads	Alignment	Compression	Decompression
2	5,956	6,903	387
4	3,050	3,605	235
8	1,650	1,989	158
16	852	1,117	114

Table 9 shows the breakdown for one- and non- aligned PE reads in each of the categories described in Section 8 for all datasets. It can be observed that the percentage of reads in non-aligned-z category is zero or close to it for all datasets. Therefore, even when both ends of a paired-end read do not align, we have sufficient evidence to orient and order reads. Note that classical alignment algorithms do not provide any information for orienting and ordering non-aligned reads.

Table 9: Percentage of one- and non- aligned paired-end reads using specialized alignment algorithm

Difference type	*CE*	*GG*	*H1*	*H2*	*H3*
One-aligned	8.17	5.87	8.21	7.94	8.84
Non-aligned-x	2.74	9.09	2.96	1.51	2.34
Non-aligned-y	3.95	0.98	0.26	0.51	0.36
Non-aligned-z	0.02	0.00	0.00	0.00	0.00

11 CONCLUSION

Owing to the large size of the read datasets produced by high-throughput sequencing instruments, transmission, storage, and archival of such datasets pose significant challenges. The magnitude of the problem grows as the sequencing technology improves. Domain-specific properties of read datasets make it difficult for general-purpose compressors to fully exploit the redundancy present in these datasets. Researchers proposed special-purpose compression algorithms, that exploit one or more of the unique properties of read datasets, to improve upon the compression efficiency of general-purpose compressors.

In this work, we developed a reference-based compressor for genomics read datasets which exploits all of the domain-specific properties of read datasets. Our compressor performs a specialized alignment of paired-end (PE) reads to standard reference sequence. It classifies PE reads based on the number of ends aligned. Finally, it uses custom compression strategies for reads in different categories. Our special-purpose compressor allows lossless transformations of PE reads in datasets. By leveraging all of these insights, it is able to significantly improve upon the compression efficiency over state-of-the-art. In addition to enhancing compression efficiency, we furnished fast and scalable parallel algorithms for compressing and decompressing read datasets.

ACKNOWLEDGMENTS

This work is supported in part by the U.S. National Science Foundation under SHF-1718479.

REFERENCES

[1] Gaëtan Benoit, Claire Lemaitre, Dominique Lavenier, Erwan Drezen, Thibault Dayris, Raluca Uricaru, and Guillaume Rizk. 2015. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC bioinformatics* 16, 1 (2015), 288.
[2] James K Bonfield and Matthew V Mahoney. 2013. Compression of FASTQ and SAM format sequencing data. *PloS one* 8, 3 (2013), e59190.
[3] Stefan Canzar and Steven L Salzberg. 2017. Short read mapping: An algorithmic tour. *Proc. IEEE* 105, 3 (2017), 436–458.
[4] Shubham Chandak, Kedar Tatwawadi, Idoia Ochoa, Mikel Hernaez, and Tsachy Weissman. 2018. SPRING: a next-generation compressor for FASTQ data. *Bioinformatics* (2018).
[5] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. 2009. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research* 38, 6 (2009), 1767–1771.
[6] Anirban Dutta, Mohammed Monzoorul Haque, Tungadri Bose, Ch V Siva K Reddy, and Sharmila S Mande. 2015. FQC: A novel approach for efficient compression, archival, and dissemination of fastq datasets. *Journal of bioinformatics and computational biology* 13, 03 (2015), 1541003.
[7] Faraz Hach, Ibrahim Numanagić, Can Alkan, and S Cenk Sahinalp. 2012. SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics* 28, 23 (2012), 3051–3057.
[8] Zhi-An Huang, Zhenkun Wen, Qingjin Deng, Ying Chu, Yiwen Sun, and Zexuan Zhu. 2017. LW-FQZip 2: a parallelized reference-based compression of FASTQ files. *BMC bioinformatics* 18, 1 (2017), 179.
[9] Daniel C Jones, Walter L Ruzzo, Xinxia Peng, and Michael G Katze. 2012. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic acids research* 40, 22 (2012), e171–e171.
[10] Carl Kingsford and Rob Patro. 2015. Reference-based compression of short-read sequences using path encoding. *Bioinformatics* 31, 12 (2015), 1920–1928.
[11] Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics* 25, 14 (2009), 1754–1760.
[12] Yuansheng Liu, Zuguo Yu, Marcel E Dinger, and Jinyan Li. 2018. Index suffix–prefix overlaps by (w, k)-minimizer to generate long contigs for reads compression. *Bioinformatics* 35, 12 (2018), 2066–2074.
[13] Ibrahim Numanagić, James K Bonfield, Faraz Hach, Jan Voges, Jörn Ostermann, Claudio Alberti, Marco Mattavelli, and S Cenk Sahinalp. 2016. Comparison of high-throughput sequencing data compression tools. *Nature Methods* (2016).
[14] Rob Patro and Carl Kingsford. 2015. Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics* 31, 17 (2015), 2770–2777.
[15] Jason A Reuter, Damek V Spacek, and Michael P Snyder. 2015. High-throughput sequencing technologies. *Molecular cell* 58, 4 (2015), 586–597.
[16] Łukasz Roguski and Sebastian Deorowicz. 2014. DSRC 2 - Industry-oriented compression of FASTQ files. *Bioinformatics* 30, 15 (2014), 2213–2215.
[17] Łukasz Roguski, Idoia Ochoa, Mikel Hernaez, and Sebastian Deorowicz. 2018. FaStore–a space-saving solution for raw sequencing data. *Bioinformatics* 1 (2018).
[18] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory* 24, 5 (1978), 530–536.