COMPRESSION OF RAW GENOMIC DATA

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Shubham Chandak

May 2021

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Tsachy Weissman, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hanlee Ji

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mary Wootters

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

With the rapid advances in genomic sequencing, the amount of genomic data being produced is growing exponentially. Several large scale sequencing projects for humans and other species are expected to further increase the volume of this data. While the initial progress was led by second generation high-throughput sequencers such as Illumina, more recently there has been increasing interest in third generation sequencers like Oxford Nanopore that enable real-time and portable sequencing of long reads. In this context, compression techniques play a crucial role in enabling efficient storage and transfer of this data. Unfortunately, the traditional general-purpose compressors like Gzip are unable to fully exploit the inherent redundancy in this data. Furthermore, in many cases the data is noisy, and it is possible to deploy lossy compression algorithms that can reduce the storage space without adverse impacts on the data quality for downstream analysis.

This thesis presents two specialized compressors for genomic data, focusing on raw genomic data which consists of sequencing reads (FASTQ format) as well as raw signal data produced by nanopore sequencing (FAST5 format). We first describe SPRING, which is an efficient compressor for unaligned single and paired-end genomic reads, supporting various lossless and lossy compression modes. Next, we discuss lossy compression of nanopore raw signal data using LFZip, which is a general-purpose lossy compressor for time series and sensor data. We also discuss the evaluation of the impact of lossy compression on the performance of downstream applications like basecalling, consensus and methylation calling.

Acknowledgements

I have had the great pleasure of being part of the Stanford community during my PhD, where I have grown so much, both at a technical and a personal level. Firstly, I wish to thank my advisor Tsachy Weissman for his incredible support throughout my stay at Stanford. He has been an exceptional mentor in research and in life, helping me discover new research opportunities and open problems, and broadening my horizons through challenging and exciting organizational, mentorship, outreach, and teaching assignments. I have always been able to rely on him as an advisor and a friend, and I still wonder at my great fortune of becoming part of his group.

I would like to thank my thesis reading committee members Hanlee Ji and Mary Wootters. Hanlee has been an outstanding co-advisor, introducing me to the field of DNA storage and providing such great guidance throughout the project. I learnt a lot from Hanlee and Mary about their respective fields, and about the value of interdisciplinary collaboration. I also wish to thank James Zou and Ayfer Özgür for being part of my exam committee and for providing useful feedback on my research. I extend my gratitude to all my teachers and professors at school, coaching institutes, at IIT Bombay, and at Stanford.

I want to thank the Electrical Engineering department for their help with the administrative support that enabled me to focus on my research.

I am thankful to all my lab members and collaborators over the years: Kedar, Billy, Pulkit, Jay, Idoia, Mikel, Qingxi, Yifan, Mohit, Jan, Reyna, Chengtao, Roshan, Matt, Srivatsan, Peter, Patrick, Dmitri, Joachim, Irena, Meltem, Jiantao, Yanjun, Yihui, Lele, Ariana, Noah and Berivan. It has been fun interacting with such excellent researchers. I extend my special thanks to Kedar for being a great friend and mentor

for so many of my projects, and for making me feel at home in the group right from my first day at Stanford.

I wish to thank my friends who have been by my side through thick and thin. My conversations with them have helped me stay grounded through this journey. Finally, my gratitude for my extended family, my brother Siddharth, and my parents cannot be expressed in words, all my achievements are built upon their love and their efforts. Thank you!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Our genome is a big part of our lives: it is the primary hereditary material which we inherit from our parents, and it affects our physical characteristics and susceptibility to various diseases. At a basic level, the genome is just a sequence of DNA bases, with each base being one of the four possibilities $\{A, C, G, T\}$. The DNA occurs in a double-stranded form, with the two strands being complementary to each other. The human genome has length around 3 billion bases spread across 23 chromosomes, and we have two copies of the genome, one from each parent.

Given the importance of the genome to human health and the understanding of life in general, there has been interest in methods to study the genome. Genome sequencing aims at reading the sequence of bases constituting the genome, often through shorter noisy substrings of the genome called *reads*. The genome is fragmented into shorter pieces which are read using a sequencer. Typically, a single position of the genome is sequenced as part of several reads, that allows us to gather useful information despite the short length of the reads and the noise in the sequencing process.

Over the past two decades, the cost of sequencing has been dropping rapidly leading to several large-scale genome sequencing projects, focusing on both humans and on other species. This has led to an explosion of data, with an estimated storage requirement of 40 exabytes predicted over the next decade (1 exabyte = 1 billion gigabytes). Thus, efficient compression of this data is critical for enabling its storage, transfer, and analysis. Given the redundancy between the reads, there is significant

potential for savings if appropriate data compression techniques are used. However, in practice, general-purpose compressors like Gzip are often used, which do not achieve the best results for genomic data with its unique statistical properties. In addition, the data is frequently noisy and further storage reduction can be achieved by using lossy compression, often without a negative impact on the data quality.

This thesis introduces specialized methods for effective compression of genomic data, with special focus on the raw data obtained from the sequencer. While typical applications ultimately process the raw data to obtain more interpretable results, the storage of raw data can become the bottleneck due to its large size. In addition, the raw data needs to be retained for long periods of time for regulatory reasons and to enable reanalysis as better analysis tools become available.

The first part of the thesis, presented in Chapter 2, concerns SPRING, which is a compressor for genomic sequencing reads focusing on short-read technologies. We discuss the algorithm in detail and analyze its performance in different modes and for different datasets. Next, Chapter 3 covers LFZip which is a general-purpose lossy compressor for time-series data with applications ranging from sensor arrays to genomic data. We discuss the algorithm inspired by a classic lossy compression framework and discuss the results across several real-life datasets. Finally, Chapter 4 applies LFZip to raw current data from nanopore sequencing. Nanopore sequencing has gained a lot of traction over the past five years and has several advantages over the older short-read technologies. We study the space savings achievable by using the lossy LFZip compression for this data, and carefully analyze the impact of lossy compression on the performance of downstream applications.

Parts of this thesis have been previously published in the following papers:

1. S. Chandak, K. Tatwawadi, I. Ochoa, M. Hernaez and T. Weissman; SPRING: A next-generation compressor for FASTQ data, *Bioinformatics*, Volume 35, Issue 15, 1 August 2019, Pages 2674–2676. [1]

2. S. Chandak, K. Tatwawadi, C. Wen, L. Wang, J.A. Ojea and T. Weissman; LFZip: Lossy compression of multivariate floating-point time series data via improved prediction, *2020 Data Compression Conference (DCC)*, Snowbird,

UT, USA, 2020, pp. 342-351. [2]

3. S. Chandak, K. Tatwawadi, S. Sridhar and T. Weissman; Impact of lossy compression of nanopore raw signal data on basecalling and consensus accuracy, *Bioinformatics*, Volume 36, Issue 22-23, 1 December 2020, Pages 5313–5321. [3]

Chapter 2

SPRING: a next-generation compressor for FASTQ data

2.1 Introduction

There has been a tremendous increase in the amount of genomic data produced in the past few years, mainly driven by the improvements in High-Throughput Sequencing (HTS) technologies and the reduced cost of sequencing a genome. A single genome sequencing experiment on humans typically results in hundreds of millions of short reads (of length 100-150bp), which are (possibly corrupted) substrings of the same underlying genome sequence. This raw sequencing data is typically stored in the FASTQ format, which consists of the reads along with the quality values which indicate the confidence in the read sequence, and read identifiers which consist of metadata related to the sequencing process. In most cases, the reads are sequenced in pairs from short fragments of the genome, resulting in paired-end FASTQ files. See Figure 2.1 for an illustration of short paired-end read sequencing. A typical FASTQ dataset for a human genome sequencing experiment requires hundreds of GBs of storage space (for a typical sequencing coverage of 30x). Due to the huge sizes involved, compression of the FASTQ files is of utmost importance for their storage and distribution.

There is significant amount of recent work on FASTQ compression [4], including SCALCE [5], Fqzcomp [6], DSRC 2 [7] and FaStore [8]. Since the reads are substrings

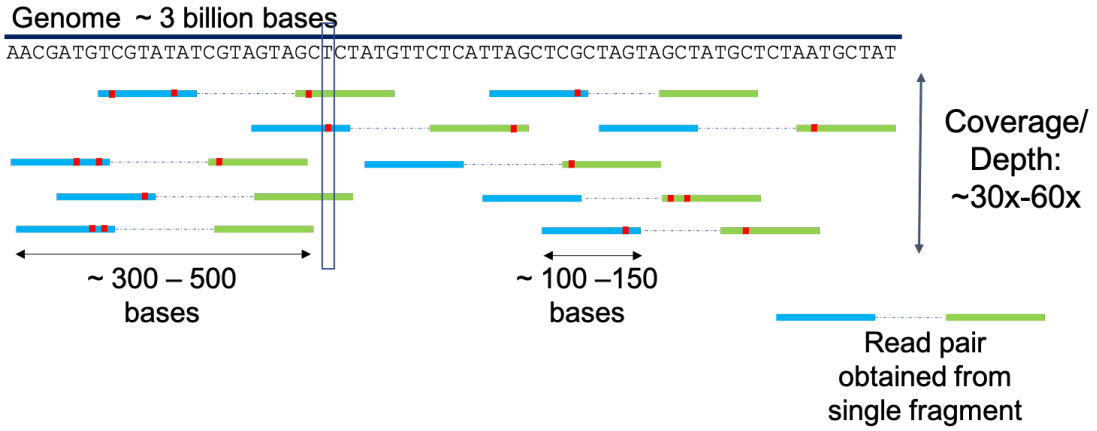


Figure 2.1: Illustration of short paired-end read sequencing.

of the underlying genome, there is much redundancy to be exploited for compression. Specialized compressors, which explicitly utilize the structure present in the reads, can achieve a compression gain of more than 10x as compared to generic universal compressors such as Gzip [4]. The quality values, on the other hand, have less structure, and thus can take up a more significant fraction of the storage space in the compressed domain. Recent work [8, 9] has shown that the quality values can be lossily compressed without adversely affecting the performance of variant calling, one of the most widely used downstream application in practice. Moreover, newer technologies such as Illumina's Novaseq are using quality values with fewer levels (4 levels instead of the previous 8 or 40 levels), hence supporting the claim that the precision in the quality values can be reduced with no impact on variant calling performance.

Although there has been a lot of work on designing FASTQ compressors, most of them lack in support of one or more crucial properties, such as support for variable length reads [8], scalability to high coverage datasets, pairing-preserving compression [7] and lossless compression [5]. Partly due to these factors, Gzip is still the prevalent FASTQ compressor, even though it provides worse compression ratios [4].

In this chapter, we present the next-generation compressor SPRING, which supports all the crucial properties, while achieving significantly better compression as compared with state-of-the-art FASTQ compressors. SPRING is also eminently practical in terms of its memory/time requirements, and supports selective access to the

compressed data.

SPRING supports the following recommended modes of FASTQ compression:

1. **Lossless mode (default):** In this mode, the FASTQ file is compressed so that it can be exactly reconstructed, i.e., the reads, quality, read identifiers and the read order information can be perfectly recovered.

2. **Recommended lossy mode:** In this mode, the information relevant for most of the genomic applications (such as alignment, assembly, variant calling, etc.) is preserved. This includes the reads along with pairing information and binned quality values. The quality values are subjected to the Illumina's standardized 8-level binning (`https://www.illumina.com/documents/products/whitepapers/whitepaper_datacompression.pdf`) before compression (Novaseq qualities are left unchanged). The read identifiers and the order of the pairs is discarded (i.e., the decompressed FASTQ file contains the read pairs in an arbitrary order). The relative ordering of the first and the second read in each pair is still preserved.

Although we advocate for these default modes, SPRING can be highly customized based on the user needs, and provides additional capabilities such as custom binning of quality values using QVZ [10] and binary thresholding.

For short reads (up to 511 bp), the read compression in SPRING is based on HARC [11], with significant improvements and added support for variable-length reads. SPRING also supports long read compression, where BSC (`https://github.com/IlyaGrebnov/libbsc/`) is used as the read compressor. Furthermore, SPRING compresses the streams in blocks, allowing for fast decompression of a subset of reads (random access). More details and results for these features are provided in the following. SPRING is open-source and available on GitHub at `https://github.com/shubhamchandak94/Spring/`.

2.2 Methods

2.2.1 FASTQ files

The FASTQ format [12] is used to represent data obtained from a sequencing experiment. This data is not aligned to a reference genome and consists of reads, quality values and read identifiers. Every read and the corresponding metadata is represented by a block of four lines. The first line is the read identifier, the second line is the read itself and the fourth line is the quality value for each base in the read. The third line contains the symbol '+' to separate the read and the quality values. This line can contain some metadata or comments, but they are rarely used and hence most compressors (including SPRING) discard them [4]. The read is a string of DNA symbols (typically A, C, G, T and N, where N represents no call). The quality value represents the confidence in each base call, encoded in ASCII using the Phred scale [13]. The read identifiers store various fields related to the sequencing process, such as lane number, instrument name, etc. For paired-end sequencing, two FASTQ files are produced, with the i^{th} read in the first file being the pair of the i^{th} read in the second file. As an example, the first read for the two files in the ERP001775 dataset is shown in Figure 2.2 (note that the read identifiers for the paired reads differ only in the last character).

2.2.2 SPRING

Figure 2.3 shows the overall compression flow for SPRING. Before describing the steps in more detail, we define some terms and discuss certain components:

- *BSC* - BSC (`https://github.com/IlyaGrebnov/libbsc/`) is a general-purpose compressor based on the Burrows-Wheeler Transform (BWT) built for achieving high compression ratios while being computationally efficient. We use BSC in a number of places in SPRING, with block length of 64 MB and `-p` (no preprocessing) flag activated.
- *Read compression* - There are two modes for read compression:

Figure 2.2: Paired-end FASTQ files in ERP001775 dataset.

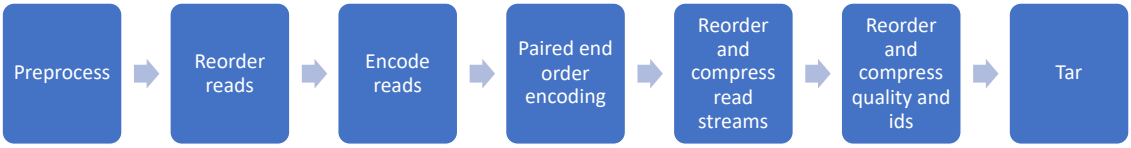


Figure 2.3: Compression flow for SPRING.

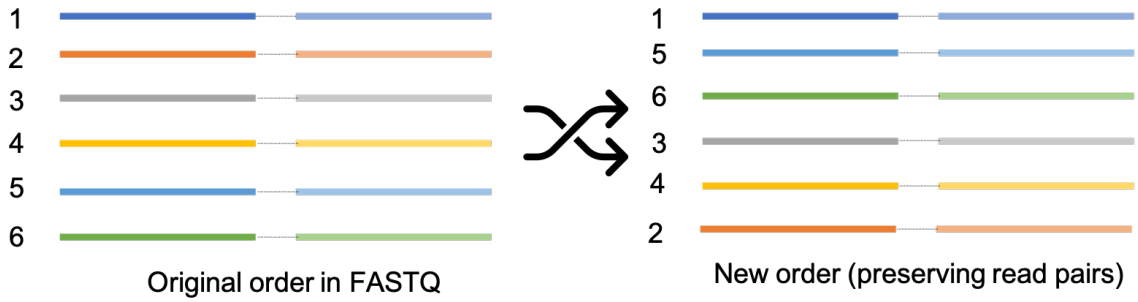


Figure 2.4: Reordering of paired reads while preserving the pairing information.

  - *Order preserving compression* - Compression preserving the order of reads in the FASTQ file. This is the default mode for SPRING.
  - *Order non-preserving (pairing only) compression* - In this mode, the order of reads in the FASTQ file is not preserved but the pairing information is preserved (see Figure 2.4). Single end FASTQ files are arbitrarily reordered in this mode. Paired end FASTQ files are reordered such that the reads in file 1 remain in file 1, reads in file 2 remain in file 2, and the paired reads still remain paired. This mode is activated by the `-r` flag.

- *Quality value compression* - SPRING uses BSC for quality value compression and allows several options for quantization:
  - *Lossless* - default mode.
  - *QVZ* - QVZ [10] models quality values as a first-order Markov process with position-dependent transition probabilities. This allows QVZ to capture both the degradation of quality values at the end of the read and the correlation between the quality values within a read. QVZ first computes statistics for this model and generates quantization codebooks using a variant of Lloyd-Max algorithm. Note that we use QVZ only for quantization of quality values, which are then compressed using BSC. SPRING supports QVZ with mean square error distortion, where the user needs to specify the desired rate in bits/quality value. It was shown in [9] that an average rate of 1 bit per quality value retains the performance of variant calling. This mode is activated by the `-q qvz qvz_rate` flag.
  - *Illumina binning* - SPRING supports Illumina's standardized 8-level binning scheme [14] for lossy compression of quality values. The Illumina binning scheme maps the 40-level quality values to 8 levels by clustering together similar values. This mode is activated by the `-q ill_bin` flag.
  - *Binary thresholding* - SPRING supports binary quantization of quality values which was shown in [8] to significantly reduce the compressed size of quality values without sacrificing variant call accuracy (for high coverage

datasets). The user needs to provide three parameters: *thr*, *high* and *low*. Quality values less than *thr* are quantized to *low* and quality values greater than or equal to *thr* are quantized to *high*. This mode is activated by the `-q binary thr high low` flag.

- *Read Identifier Compression* - As the read identifiers consist of heterogeneous but structured data, SPRING uses a token-based approach for their compression. The read identifiers are split into tokens and each token is encoded separately. For the numeric tokens, SPRING uses delta encoding if the difference from the previous value is small, otherwise it stores the token value as it is. For string type tokens, SPRING uses a special symbol to denote a perfect match with the previous value, otherwise the full string is stored. Finally these streams are compressed using an adaptive arithmetic encoder. The read identifier compression in SPRING is based on Samcomp [6] and GeneComp [15].

  For paired-end datasets, typically the corresponding identifiers in the two files differ only in a single character. During the preprocessing stage, we check if the identifiers have this structure. In that case, identifiers for only one of the files are compressed and those for the other file are reconstructed during the decompression.

- *Short read mode* - This mode supports short reads with read lengths up to 511. The short read compression in SPRING is based on HARC [11], with added support for paired end reads and several other improvements (discussed later in more detail). This mode is optimized for relatively accurate short reads containing substitution errors and is the default mode for SPRING.

- *Long read mode* - This mode supports long reads with read lengths up to 4.2 billion. Here we use BSC for read compression. This mode is activated by the `-l` flag and is always order preserving. Since the short read mode is designed for low error rates with most errors being subsitutions, the long read mode is also recommended for short read datasets with large number of indel errors.

Preprocess

In the long read mode, the reads, quality values and read identifiers are separated and compressed in blocks (reads and quality values using BSC, identifiers using specialized identifier compressor described above). By default, the block length for long reads is set to 10,000 reads. The read lengths are also stored as 32-bit integers in a separate stream which is compressed using BSC. Preprocessing is directly followed by the Tar stage for long reads.

In the order preserving mode, the quality values and read identifiers are compressed in blocks (quality values using BSC, identifiers using specialized identifier compressor). QVZ quantization is applied before quality compression if the corresponding flag is specified. By default the block length for short reads is set to 256,000 reads (see Section 2.4.8 for impact of block length on compression). The reads are written to temporary files after separating out the reads containing the character 'N'. The reads containing 'N' are considered directly in the "Encode reads" stage.

In the order non-preserving mode, the quality values and read identifiers are written to temporary files. The reads are handled exactly as in the order preserving mode.

If the Illumina binning or binary thresholding flag is activated, the qualities are binned before compression/writing to temporary file.

Reorder reads

This step in read compression is based on HARC [11], with several extensions and improvements. Here, we will provide a brief overview of the step, more details and parameters can be found in [11]. In this step, SPRING reorders the reads so that they are approximately ordered according to their position in the genome. The reordering is done in an iterative manner: given the current read, SPRING tries to find a read which matches the prefix or the suffix of the current read with a small Hamming distance. To do this efficiently, a hash table is used which indexes the reads according to certain substrings of the read. SPRING makes the following improvements to this stage:

- While HARC searched for matching reads in only one direction (matching the suffix of the current read), SPRING looks for matches in both directions. This boosts read compression by 5-10% on most datasets (see Section 2.4.10).
- While HARC only supported fixed length reads of maximum length 255, SPRING adds support to variable length short reads of maximum length 511. For this, SPRING stores an array containing the read lengths, which is used to ensure that the Hamming distance between reads of different lengths is computed correctly.
- We observed that most of the time in the reordering stage is spent on a small fraction of remaining reads and the attempts to find matches to these reads usually fails. To save time in this step, SPRING introduces early stopping to this stage. Each thread maintains the fraction of unmatched reads in the last 1 million reads and stops looking for matches once this fraction crosses a certain threshold (50% by default). Since this stage is the most time-consuming step in SPRING compression, early stopping can reduce compression times by as much as 20% without affecting the compression ratio (see Section 2.4.10).

Encode reads

In this step, the sequence of reordered reads is used to obtain a majority-based reference sequence. The reference sequence is then used to encode the reordered reads. The final encoding includes the reference sequence, the positions of the reads in the reference sequence, and the mismatches of reads with respect to the reference sequence. An index mapping the reordered reads to their position in the original FASTQ file is also stored. This step is almost unchanged from HARC [11] and more details can be found there. The only major addition in SPRING is the support for variable-length reads of lengths up to 511.

This stage produces a majority-based reference sequence and encoded streams for reads aligned to the reference. A small fraction of reads usually remains unaligned to the reference to the reference and are stored separately. However, the encoded streams do not correspond to the original order of reads in the FASTQ file. Furthermore, the

reordering and encoding stages from HARC consider the paired end FASTQ files as a single end FASTQ file obtained by concatenating the two files. Thus, for both the order preserving and order non-preserving modes, we need to transform these streams using the information in the index mapping the reordered reads to their position in the original file. This is done in the next two steps.

Paired end order encoding

This step is used only in the order non-preserving mode. Here we generate a new ordering of the reads which preserves the pairing information while achieving the optimal compression. This step generates an index mapping the reordered reads to their position in the new ordering. The reads in file 1 are kept in the same order as obtained after the previous stage (Encode reads), i.e., the reads in file 1 are sorted according to their position in the majority-based reference. This allows us to store the positions of these reads in the majority-based reference using delta-coding leading to improved compression. The ordering of the reads in file 2 is automatically determined by the ordering of reads in file 1 (since pairing information is preserved).

For single end files (in the order non-preserving mode), the reads are kept in the same order as obtained after the encoding stage (i.e., sorted according to their position in the majority-based reference).

Reorder and compress read streams

In this step, the final encoded streams are generated and compressed in blocks using BSC. For this, first the streams generated by the encoding stage are loaded into the memory. These are then reordered according to the mode. In the order preserving mode, the streams are ordered according to the original order of reads in the FASTQ files. In the order non-preserving mode, the streams are ordered according to the new order generated in the paired end order encoding step. The final streams are described below:

- *seq* - stores the majority-based reference sequence. This is packed into a 2 bits/base representation before compression.

- *flag* - indicates whether the reads are aligned or not as well as the distance between them on the reference.
  - 0 - Both reads aligned and gap between alignment positions is $< 32{,}767$ (for single end datasets, flag 0 means that the read is aligned).
  - 1 - Both reads aligned and gap between alignment positions is $\geq 32{,}767$.
  - 2 - Both reads unaligned (for single end datasets, flag 2 means that the read is unaligned).
  - 3 - read 1 of pair aligned, read 2 unaligned.
  - 4 - read 1 of pair unaligned, read 2 aligned.
- *pos* - in the order preserving mode, stores the position of the first read of the pair (and possibly the second read) on the reference using 8 bytes. If flag is 0 or 3, only the position of the first read is stored. If flag is 1, positions of both the first and the second reads are stored. If flag is 2, nothing is stored.

  In the order non-preserving mode, the position of the first read of the pair is stored as the difference from the first read of the previous pair (except for the first pair in the block). Note that the difference is always positive because of the way the new order is defined in the paired end order encoding step. This difference is stored as a 2 byte unsigned integer as long as it is $< 65{,}535$. Otherwise we store 65,535 using 2 bytes followed by the actual difference using 8 bytes. Storing differences rather than the absolute position allows SPRING to achieve significantly better compression in the order non-preserving mode.
- *pos_pair* - for paired end datasets, store the gap between the paired reads on the reference using a 16 bit signed integer when the flag is 0. Since the paired reads are sequenced from nearby portions of the genome (paired reads are typically separated by 50-250 bases), they are likely to appear close in the reference. Using a separate stream for the gap between the paired reads allows us to exploit this fact.

- *noise* - store the noisy bases in the aligned reads with respect to the reference. The encoding depends on both the base in the reference and in the read, allowing us to exploit the fact that certain errors are more likely in Illumina sequencing. For example, the most probable transitions for each reference symbol are encoded as 0, next most probable transitions as 1 and so on. This leads to more 0's in the encoded stream leading to better compression. A newline character separates the noise for consecutive reads.

- *noisepos* - stores the position of the noisy bases encoded in the *noise* stream. These are delta encoded to exploit the fact that most sequencing errors occur towards the end of the read. The delta coded noise positions are stored as 16 bit unsigned integers.

- *RC* - store the orientation (forward/reverse) of aligned reads with respect to the reference. If flag is 0, this does not store the orientation of the second read in the pair (see *RC_pair* stream).

- *RC_pair* - for paired end datasets, store the relative orientation of the second read with respect to the first read when the flag is 0. If the paired reads have opposite orientation, store 0, otherwise store 1. Since the paired end reads have opposite orientation of the genome, we expect to get mostly 0's in this stream and hence this stream is highly compressible.

- *unaligned* - stores the unaligned reads without any encoding.

- *length* - store the read lengths as 16 bit unsigned integers.

Reorder and compress quality and ids

This step is used only in the order non-preserving mode. Here the quality and ids are reordered to match the new ordering of the reads. After reordering, they are compressed in blocks (as done in preprocess stage for the order preserving mode). To reduce the memory consumption during reordering, SPRING makes multiple passes over the quality and ids. In each pass, a subset of quality and ids are loaded into

memory and these are compressed according to the new ordering. If QVZ is being used for quality value quantization, it is applied before compression. Note that Illumina binning and binary thresholding of quality values are already applied in the preprocessing step, so they are not required in this step. In case the qualities and/or identifiers are not to be preserved, this step simply ignores them.

Tar

All the compressed streams are converted to a tar archive at the end.

Decompression

During decompression, first the *seq* stream is decompressed (not applicable for long read mode). Then, multiple threads decompress the blocks in parallel which are then written to the output files by the master thread. SPRING supports decompression of a subset of reads by specifying the `--decompress-range` flag. In this case, the entire *seq* stream is decompressed and then only the blocks corresponding to the desired range of reads are decompressed.

2.3 Main results

The proposed algorithm, SPRING, was tested on a variety of datasets and compared to various algorithms. For lossless compression, we compare SPRING with pigz (`https://zlib.net/pigz/`), FaStore [8] and DSRC 2 [7]. pigz (parallelized Gzip) was chosen as it is currently the standard FASTQ compressor. FaStore does not preserve the order of the reads and hence is not lossless in general. However, for these datasets, the original order can be recovered from the read identifiers since they are sequentially ordered. For the recommended lossy mode, we compare SPRING with FaStore. The compression script for FaStore was modified so that the information retained in this mode is the same for SPRING and FaStore (details on GitHub). For both modes, we tested both the default and fast mode of FaStore. While several tools such as SCALCE [5] and Fqzcomp [6] provide much better compression

Dataset	Genome length (Mb)	Read length	#reads (M)	Coverage	PE/SE	Technology	Accession no.
E. coli	4.6	301	1.3	85	PE	MiSeq	SRR1770413
P. aeruginosa	6	100	3.3	50	PE	GAIIx	SRR554369
S. cerevisiae	12.1	63, 75	30	175	PE	GAII	SRR327342
T. cacao	350	74	69	15	SE	GAIIx	SRR870667_2
PhiX	0.0054	100	200	3.7×10^6	PE	NovaSeq	PhiX
Metagenomic	-	100	72	-	PE	HiSeq 2000	ERR532393
H. sapiens 1	3137	100	48.9	1.6	PE	GAII	SRR062634
H. sapiens 2	3137	101	879	28	PE	HiSeq 2000	ERP001775
H. sapiens 3	3137	147	540	25	PE	NovaSeq	NA12878 Rep 1, Lane 1
H. sapiens 4	3137	147	2173	100	PE	NovaSeq	NA12878 Rep 1 & 2

Table 2.1: Short read datasets used for evaluation. PE denotes paired-end, SE denotes single-end. For SRR327342, the read length of the first read in each pair is 63, and that of the second read is 75. Instructions for obtaining these datasets are available on GitHub.

than pigz, we decided to test only FaStore and DSRC 2 since they represent the state-of-the-art in terms of compression ratio and compression speed, respectively [8]. Moreover, the fast mode of FaStore still achieves better compression than the other tools, while achieving similar or faster compression speed.

All the experiments were run on a server with a 40-core Intel(R) Xeon(R) 2.20 GHz processor, 258 GB of RAM, 7.3 TB disk space and Ubuntu 18.04. All tools were run with 8 threads. 1 MB denotes 10^6 bytes and 1 GB denotes 10^9 bytes throughout the paper. Instructions for installing and using SPRING and other tools along with the commands used for the experiments are available on GitHub.

The datasets (listed in Table 2.1) include viral, bacterial, metagenomic, plant and human sequencing data and cover a range of coverages, Illumina sequencing technologies, and read lengths. Some of these datasets are part of a compilation by MPEG HTS compression working group for benchmarking purposes [4]. The human NovaSeq datasets were obtained from Illumina BaseSpace public data and consisted of variable-length ($\approx$ 150bp) paired-end reads along with 4-level quality values. These were trimmed to 147bp as FaStore does not support variable-length reads. Results for SPRING for the original variable-length datasets are in Section 2.4.5. All the datasets used in this chapter are publicly available and links to these are available on GitHub.

Dataset	Uncompressed size	pigz	DSRC 2	FaStore (fast)	FaStore	SPRING	Improvement over FaStore
E. coli	827	253	189	-	-	**106**	-
P. aeruginosa	768	279	198	142	145	**115**	1.26x
S. cerevisiae	5,986	2,062	1,507	-	-	**954**	-
T. cacao	13,847	4,926	3,540	2,755	2,714	**2,444**	1.11x
Metagenomic	19,284	6,911	5,155	3,628	3,602	**3,206**	1.12x
PhiX	50,090	6,402	6,594	1,552	1,457	**1,420**	1.03x
H. sapiens 1	12,861	3,920	2,702	2,293	2,299	**2,118**	1.09x
H. sapiens 2	227,246	74,250	52,049	36,042	35,662	**28,901**	1.23x
H. sapiens 3	195,748	36,131	26,520	11,380	11,101	**6,971**	1.59x
H. sapiens 4	787,616	144,927	106,665	35,129	33,734	**25,883**	1.30x

Table 2.2: Sizes in MB for lossless compression. FaStore wasn't run on *S. cerevisiae* since it does not support variable length reads. On *E. coli*, FaStore exited with a segmentation fault. Best results are boldfaced.

Dataset	Uncompressed size	FaStore (fast)	FaStore	SPRING	Improvement over FaStore
E. coli	827	-	-	**63**	-
P. aeruginosa	768	83	88	**62**	1.41x
S. cerevisiae	5,986	-	-	**366**	-
T. cacao	13,847	1,339	1,300	**1,215**	1.07x
Metagenomic	19,284	1,937	1,935	**1,736**	1.11x
PhiX	50,090	1,226	**1,099**	1,160	0.95x
H. sapiens 1	12,861	1,244	1,251	**1,223**	1.02x
H. sapiens 2	227,246	17,846	17,417	**13,460**	1.29x
H. sapiens 3	195,748	10,246	9,927	**5,657**	1.75x
H. sapiens 4	787,616	30,379	28,846	**20,316**	1.42x

Table 2.3: Sizes in MB for recommended lossy compression. FaStore wasn't run on *S. cerevisiae* since it does not support variable length reads. On *E. coli*, FaStore exited with a segmentation fault. Best results are boldfaced.

Tables 2.2 and 2.3 show the compression results for the lossless and recommended lossy modes, respectively. We see that SPRING consistently achieves the best compression ratios for both modes across the selected datasets, except for lossy compression of the extremely high coverage (3.7×10^6x) *PhiX* dataset. For the 28x human dataset (*H. sapiens 2*) from the Platinum Genomes Project (ERP001775) [16], SPRING achieves 1.2-1.3x better compression than FaStore. The space required for the recommended lossy mode is less than half of the lossless mode, primarily due to Illumina binning of quality values (see Section 2.4.1).

The improvement is even more significant for the NovaSeq datasets, with close to 1.75x improvement for the recommended lossy mode on the 25x-coverage dataset (*H. sapiens 3*). For the 100x NovaSeq dataset (*H. sapiens 4*), SPRING can save around 8 GB ($\approx 25\%$) storage space as compared to FaStore in both modes. The improvement provided by SPRING is not as significant for extremely high (*PhiX*) or low (*H. sapiens* 1) coverages, but these cases are less common in practice. The difference between HiSeq 2000 and NovaSeq datasets, and the contribution of reads, quality values and read identifiers to the compressed sizes are discussed further in Section 2.4.1. Finally, we observe that FaStore and SPRING achieve close to 5x better compression than pigz and 2-3x better compression than DSRC 2.

Dataset	Lossless					Recommended lossy		
	pigz	DSRC 2	FaStore (fast)	FaStore	SPRING	FaStore (fast)	FaStore	SPRING
E. coli	10s	2s	-	-	41s	-	-	41s
P. aeruginosa	31s	4s	35s	2m2s	23s	28s	1m50s	27s
S. cerevisiae	1m17s	25s	-	-	3m3s	-	-	2m55s
T. cacao	3m	1m10s	5m12s	18m	9m	3m30s	15m	9m
Metagenomic	4m38s	1m27s	7m	17m	10m	5m	14m	10m
PhiX	6m	2m8s	13m	30m	14m	11m	25m	17m
H. sapiens 1	2m37s	36s	4m37s	25m	11m	3m54s	24m	11m
H. sapiens 2	49m	13m	1h19m	3h35m	2h30m	1h	3h9m	2h32m
H. sapiens 3	33m	9m	58m	2h36m	2h	53m	2h28m	2h13m
H. sapiens 4	2h17m	43m	4h10m	9h51m	6h39m	3h50m	8h52m	7h33m

Table 2.4: Compression times. All tools were run with 8 threads.

Tables 2.4 and 2.5 contain the compression times and RAM usage, respectively,

Dataset	Lossless					Recommended lossy		
	pigz	DSRC 2	FaStore (fast)	FaStore	SPRING	FaStore (fast)	FaStore	SPRING
E. coli	0.008	0.13	-	-	1.4	-	-	1.1
P. aeruginosa	0.008	0.13	2.3	2.3	1.5	2.1	2.1	0.84
S. cerevisiae	0.008	0.13	-	-	2.3	-	-	2.3
T. cacao	0.008	0.13	4.2	4.1	3.3	3.4	3.6	3.7
Metagenomic	0.008	0.13	11	11	3.6	9.3	9.2	5.0
PhiX	0.008	0.12	25	26	18	20	24	21
H. sapiens 1	0.008	0.18	17	18	4.9	13	14	5.3
H. sapiens 2	0.008	0.42	35	31	45	25	26	45
H. sapiens 3	0.008	0.13	40	41	32	38	32	31
H. sapiens 4	0.008	0.15	158	137	119	145	122	119

Table 2.5: Compression memory (RAM) in GB. All tools were run with 8 threads.

for both modes. We observe that pigz and DSRC 2 require significantly lesser computational resources at the cost of worse compression ratios. FaStore (fast) is more than twice as fast as FaStore, while providing similar compression ratios. SPRING is competitive in terms of compression time and memory, requiring less time and memory than FaStore in most cases. SPRING is slower in the recommended lossy mode because of the additional step of reordering qualities and identifiers according to the new order of the reads.

The high memory consumption of SPRING is primarily due to the read reordering step, where SPRING loads all the reads and two hash tables into memory. The previous work on HARC [11] discusses a strategy for reducing the memory consumption by splitting the FASTQ file into multiple parts and applying the compressor independently on each part. Since the memory consumption for SPRING/HARC is linear in the number of reads, this strategy can reduce the memory consumption significantly at the cost of worse read compression.

To achieve better read compression, SPRING also devotes more time to read compression stages (reorder reads and encode reads). Together, these two stages take about 4 hours (out of total 6h39m) for the lossless compression of 100x *H. sapiens 4* dataset. We note here that SPRING introduces early stopping to the reordering stage, which reduces the overall compression time by up to 20% (see Section 2.4.10).

Further improvements in compression time can be achieved by using more threads (see Section 2.4.7).

Dataset	Lossless					Recommended lossy		
	pigz	DSRC 2	FaStore (fast)	FaStore	SPRING	FaStore (fast)	FaStore	SPRING
E. coli	3s	2s	-	-	17s	-	-	15s
P. aeruginosa	4s	2s	12s	18s	9s	7s	12s	7s
S. cerevisiae	27s	10s	-	-	1m	-	-	43s
T. cacao	1m13s	23s	2m5s	2m14s	2m20s	1m9s	1m11s	1m46s
Metagenomic	1m46s	37s	2m42s	3m	3m18s	1m21s	1m36s	2m29s
PhiX	2m23s	39s	3m3s	3m47s	5m32s	2m33s	2m11s	5m34s
H. sapiens 1	1m	18s	1m27s	1m39s	2m25s	58s	59s	2m
H. sapiens 2	20m	14m	24m	25m	38m	15m	16m	28m
H. sapiens 3	11m	9m	11m	12m	26m	9m	10m	22m
H. sapiens 4	1h21m	41m	40m	45m	1h47m	32m	36m	1h37m

Table 2.6: Decompression times. All tools were run with 8 threads.

Dataset	Lossless					Recommended lossy		
	pigz	DSRC 2	FaStore (fast)	FaStore	SPRING	FaStore (fast)	FaStore	SPRING
E. coli	0.003	0.23	-	-	1.7	-	-	1.7
P. aeruginosa	0.003	0.24	0.78	0.8	1.7	0.53	0.61	1.7
S. cerevisiae	0.003	0.43	-	-	2.2	-	-	1.9
T. cacao	0.003	0.29	1.7	2.3	2.1	1.2	1.5	1.7
Metagenomic	0.003	0.29	1.9	1.9	2.6	1.3	1.4	3.1
PhiX	0.003	0.33	19	16	2.3	15	13	2.3
H. sapiens 1	0.003	0.30	2	1.7	3.2	1.4	1.3	3.7
H. sapiens 2	0.003	0.42	26	19	5.5	21	15	5.5
H. sapiens 3	0.003	0.34	39	23	6.1	30	17	6.3
H. sapiens 4	0.003	0.36	141	85	6.6	110	81	6.7

Table 2.7: Decompression memory (RAM) in GB. All tools were run with 8 threads.

Tables 2.6 and 2.7 contain the decompression times and RAM usage, respectively, for both modes. SPRING achieves reasonably fast decompression, while using much less memory as compared to FaStore. By using more threads, SPRING can achieve faster decompression at the cost of higher memory usage (see Section 2.4.7). SPRING also supports the ability to decompress a subset of reads without needing to decompress the whole file (see Section 2.4.4).

2.4 Additional results

Unless otherwise specified, all tools were run with 8 threads.

2.4.1 Field-wise compression results

Tables 2.8 and 2.9 provide the field-wise compression results for lossless and recommended lossy mode, respectively. Since pigz does not provide field-wise compression results, it is not included in these tables. We also exclude FaStore (fast) since it is very similar to FaStore in terms of compression results (the two modes differ only in read compression). Note that the sizes for SPRING are before the Tar step which adds a small overhead. Recall that *H. sapiens 2* is 28x dataset sequenced on HiSeq 2000, while *H. sapiens 3* and *H. sapiens 4* are sequenced on NovaSeq, with coverages 25x and 100x, respectively.

Dataset	Tool	Reads	Quality	Identifier
H. Sapiens 2	DSRC 2	22,188	27,810	2,051
	FaStore	6,968	24,868	3,826
	SPRING	4,253	23,774	858
H. Sapiens 3	DSRC 2	19,845	4,576	2,098
	FaStore	6,152	3,789	1,160
	SPRING	3,040	3,630	292
H. Sapiens 4	DSRC 2	79,850	18,346	8,468
	FaStore	13,741	15,178	4,815
	SPRING	10,125	14,553	1,165

Table 2.8: Sizes (in MB) of individual fields for lossless compression.

From Table 2.8, we see that FaStore and SPRING provide significant improvement in read compression over DSRC 2, while the improvement in quality compression is smaller. Since FaStore reorders the reads even in its lossless mode, the read order information is effectively preserved in the identifiers. Due to this, the identifiers take much larger size for FaStore as compared to SPRING. Note that SPRING takes less space for read compression than FaStore even though it stores information about the read order in the read field. SPRING achieves slightly better quality compression than

Dataset	Tool	Reads	Quality	Identifier
H. Sapiens 2	FaStore	6,917	10,500	0
	SPRING	2,553	10,892	0
H. Sapiens 3	FaStore	6,138	3,789	0
	SPRING	2,022	3,625	0
H. Sapiens 4	FaStore	13,668	15,178	0
	SPRING	5,722	14,558	0

Table 2.9: Sizes (in MB) of individual fields for recommended lossy compression.

FaStore due to the use of BSC instead of QVZ. Comparing the HiSeq 2000 dataset (*H. sapiens 2*) to the other two NovaSeq datasets, we observe that the quality takes up a much smaller fraction of the total size for NovaSeq datasets which have only 4 quality levels. For the NovaSeq data, the size required for reads is quite comparable to that required for qualities. Due to this, SPRING provides greater improvement for these datasets.

In Table 2.9, both FaStore and SPRING reorder the reads and only retain the pairing information. We see that SPRING requires 2.5-3x less space for compressing the reads in this mode. Comparing with the lossless mode (Table 2.8), we see that Illumina binning reduces the space needed to store the quality values by more than 2x for *H. sapiens 2*. For SPRING, storing only the pairing information rather than the complete order of reads boosts the read compression by a factor of 1.5-1.7, with more improvement for higher coverage datasets.

2.4.2 Comparison with alignment + SAM compression

In applications such as sequencing of a new organism or metagenomics, a reference is typically not available and hence reference-free compression of FASTQ files is important. Even if a reference is available, the FASTQ file is typically retained (at least temporarily) and again FASTQ compression becomes necessary. Since the reference and the FASTQ file are usually obtained from different individuals, using a reference-free compressor like SPRING can be beneficial since it is more robust to variations between individuals. To understand this better, we compared SPRING to

reference-based alignment. We first used BWA-MEM [17] to align the *H. sapiens 3* dataset to the hg19 reference. Then we removed some irrelevant fields from the SAM file (MAPQ, RNEXT, PNEXT, TLEN and optional fields), since these are not used to compress data in the FASTQ file. Finally, we used CRAM v3 (from SAMtools) to compress the SAM file, both before and after sorting according to the genome position. The commands used for these operations are available on GitHub. The parameters were chosen to achieve best compression.

The compressed sizes for the unsorted and sorted SAM files are 7,644 MB and 7,793 MB, respectively. The compressed size for the sorted SAM file with the reference embedded in the CRAM file is 8,488 MB. In comparison, SPRING achieves 6,971 MB without using any external reference. While the CRAM compression step is quite fast (around 25m), the alignment took around 8 hours. SPRING needs 2 hours for compressing this dataset. Thus, we see that directly compressing FASTQ files can be advantageous even when a reference is available. For species with larger variation between individuals, SPRING can provide even greater improvements over alignment + SAM compression.

2.4.3 Long read compression

Accession no.	Species	Genome length (Mb)	Maximum read length	#reads (M)	Coverage	Technology
SRR1284073	*E. coli*	4.6	49424	0.65	140	PacBio
ERR637420	*E. coli*	4.6	47422	0.08	86	Oxford Nanopore MinION

Table 2.10: Long read datasets used for evaluation. Both datasets are single end.

Accession no.	Uncompressed	pigz	SPRING
SRR1284073	1,304	546	420
ERR637420	264	120	94

Table 2.11: Sizes in MB for long read compression.

We compared the long read compression (lossless) mode with pigz, since DSRC

2 and FaStore do not support long reads. We evaluated the tools on two datasets (Table 2.10) from the most popular long read platforms. The compression results are shown in Table 2.11. We observe that SPRING achieves better compression than pigz on these datasets, but the improvement is not as pronounced as that for short read datasets. This is because SPRING uses the general-purpose compressor BSC for long read compression rather than the specialized compression method employed for short reads, and hence it is unable to exploit much of the redundancy in the reads. Building a specialized read compressor for long reads is part of future work.

2.4.4 Decompressing subset of reads

SPRING allows decompression of a subset of reads by specifying a range of reads to decompress. For paired end files, the parameters refer to read pairs rather than reads. Table 2.12 shows the times needed to decompress 1M, 10M and 100M read pairs from losslessly compressed *H. sapiens 3*, along with the time needed to decompress all read pairs ($\approx$ 270M). The results were obtained by using the `--decompress-range` flag. We see that SPRING allows fast decompression of small subsets of reads, with a slight constant overhead due to decompression of the *seq* stream.

Start pair	End pair	Number of pairs	Decompression time
100M	101M	1M	1m13s
100M	110M	10M	1m48s
100M	200M	100M	9m4s
-	-	270M	22m

Table 2.12: Time required to decompress subset of read pairs for *H. sapiens 3*. Last row represents decompression of entire file.

2.4.5 Results for variable length short reads

Table 2.13 contains compression results for NovaSeq variable length reads. FaStore does not support variable length reads, so only SPRING, pigz and DSRC 2 were

tested. On these datasets, SPRING provides 3-5x better compression than pigz and DSRC2.

Sample	NA12878 Rep 1, Lane 1 (original)	NA12878 Rep 1 & 2 (original)
Organism	*H. sapiens*	*H. sapiens*
Technology	NovaSeq	NovaSeq
Coverage	26x	105x
Maximum Read length	151	151
Uncompressed Size	205,386	826,117
Lossless		
pigz	38,007	152,243
DSRC 2	28,448	114,393
SPRING	7,565	29,020
Recommended lossy		
SPRING	6,193	22,954

Table 2.13: Compression sizes in MB for the variable-length NovaSeq datasets. Only tools supporting variable length reads were tested.

2.4.6 Quality value lossy compression modes

Mode	Parameters	Compressed quality size
Lossless	-	23,774
Illumina binning	-	10,892
QVZ	Rate = 1 bit/quality value	7,237
Binary thresholding	thr=20, high=40, low=6	1,034

Table 2.14: Sizes in MB for different quality value compression modes for *H. sapiens 2*.

While the recommended lossy mode for SPRING uses Illumina 8-level binning for quality values, SPRING supports two other schemes for lossy compression of quality values. Table 2.14 shows the compressed sizes for these schemes for *H. sapiens 2* (HiSeq 2000, 28x). In previous works [8, 9], all these were shown to have no

detrimental effect on variant calling (binary thresholding can hurt variant calling for low coverage datasets). We see that binary thresholding can reduce the space needed by qualities significantly. Since QVZ uses an optimized context-dependent quantizer dependent on the input data, it is slightly slower, but provides more flexibility in terms of the desired rate and should provide lower distortion at the same rate than other schemes.

2.4.7 Impact of number of threads

Number of threads	Compressed size (MB)	Compression time	Compression memory (GB)	Decompression time	Decompression memory (GB)
4	6,977	3h26m	31	42m	4.6
8	6,971	2h	32	26m	6.1
16	6,960	1h20m	32	18m	9.3
32	6,968	1h6m	32	13m	15

Table 2.15: Impact of number of threads on compressed size and time/memory consumption for lossless compression of *H. sapiens 3*.

Table 2.15 shows the compressed sizes and computational requirements for lossless compression *H. sapiens 3* dataset for three values of number of threads. The results were obtained by using the `-t` flag. We observe that the compression and decompression times improve as the number of threads increase. For very high number of threads, the disk I/O becomes the bottleneck leading to diminishing returns. The impact of increasing the number of threads on the compressed size and compression memory usage is negligible. The decompression memory increases with the number of threads because more blocks are now decompressed in parallel.

2.4.8 Impact of block size

SPRING compresses the streams in blocks to allow random access and efficient decompression. The number of reads (read pairs for PE datasets) per block is set to 256,000 by default (for short reads). Table 2.16 shows the compressed sizes and computational requirements for lossless compression *H. sapiens 3* dataset for three values of block

Block size	Compressed size (MB)	Compression time	Compression memory (GB)	Decompression time	Decompression memory (GB)
128,000	7,011	1h59m	31	26m	5.5
256,000	6,971	2h	32	26m	6.1
512,000	6,950	1h56m	32	24m	8.9

Table 2.16: Impact of block size on compressed size and time/memory consumption for lossless compression of *H. sapiens 3*.

size. The results were obtained by modifying the parameter `NUM_READS_PER_BLOCK` in `src/params.h`. We observe that using higher block sizes yields slightly better compression but needs more memory during decompression. The impact on compression and decompression times is negligible for this range of block sizes.

2.4.9 Impact of read reordering on ID compression

While read identifiers are not used in most downstream applications, some applications like Picard MarkDuplicates (`http://broadinstitute.github.io/picard/`) might use it. In such cases, we recommend that SPRING be used without the `-r` flag which allows read reordering (with pairing information preserved). When the `-r` flag is specified, the read identifiers are reordered and then compressed. Due to this, the consecutive read identifiers now contain large differences, leading to poor compression. Even though the size needed for the reads reduces, the increase in the read identifier size slightly outweighs this reduction. Table 2.17 shows this for *H. sapiens 3* dataset.

Flag	Read	Read identifier	Read + Read identifier
`-r` used	2,020	1,371	3,391
`-r` not used	3,040	292	3,332

Table 2.17: Impact of using `-r` flag on read and read identifier compression for *H. sapiens 3*. Sizes are in MB.

2.4.10 Improvements in reordering stage

Here we discuss the impact of two improvements made in SPRING to the reordering stage of HARC.

Searching for matches in both directions

Mode	Compressed size of reads (MB)	Compression time	Compression memory (GB)
Search in one direction	3,251	1h59m	32
Search in both directions	3,040	2h	32

Table 2.18: Impact of bidirectional search on compressed size and time/memory consumption for lossless compression of *H. sapiens 3*.

While HARC searched for matching reads in only one direction (matching the suffix of the current read), SPRING looks for matches in both directions. Table 2.18 shows the impact of this on read compression for lossless compression of *H. sapiens 3* dataset (note that the reported times include time for quality and identifier compression). The results for the first row were obtained by replacing `src/reorder.h` in the SPRING repository by `src/old_src/reorder_1d/reorder.h`. We observe that bidirectional search improves the read compression by around 6% without significantly affecting the compression time/memory.

Early stopping

Mode	Compressed size of reads (MB)	Compression time	Compression memory (GB)
No early stopping	10,107	9h15m	119
Early stopping	10,125	6h39m	119

Table 2.19: Impact of early stopping on compressed size and time/memory consumption for lossless compression of *H. sapiens 4*.

SPRING maintains the fraction of unmatched reads in the last 1 million reads and stops looking for matches once this fraction crosses a certain threshold (50% by default). The maximum impact of this step is on the largest dataset *H. sapiens 4*. Table 2.19 shows results for lossless compression of *H. sapiens 4* where the reported times includes the time for quality and identifier compression. The results for the first row were obtained by setting `STOP_CRITERIA_REORDER` to 1.0 in `src/params.h`. We see that early stopping reduces the compression time by more than 20% for this dataset while having negligible impact on compressed size and memory usage.

2.5 Conclusions

This chapter presented the FASTQ compressor SPRING, which outperforms existing tools, offering 1.3x-1.8x improvement in compression over the next best performing tool on data sequenced on Illumina's latest sequencer, NovaSeq. SPRING supports a wide variety of modes and features and is competitive in terms of computational requirements. We note that there have been some compressors developed after SPRING, notably PgRC [18] and FQSqueezer [19], sometimes obtaining better compression results than SPRING at the cost of added computation. These compressors support many of the same compression modes as SPRING, and PgRC uses a similar approximate assembly-based approach.

The streams generated by SPRING can be easily transformed to streams compatible with the upcoming standard developed by the MPEG-G group for genomic information representation [20] and work on integration of SPRING into the standard is currently being pursued as part of the genie project (`https://github.com/mitogen/genie/`). We are also developing specialized read compressors for long read technologies that can effectively handle insertion and deletion errors.

Chapter 3

LFZip: Lossy compression of multivariate floating-point time series data via improved prediction

3.1 Introduction

With the rapid increase in smart machines, IoT devices and sensors collecting and transmitting large volumes of measurement data, it has become essential to consider data compression strategies to reduce the transmission volume. This chapter focuses on the problem of multivariate time series compression, which arises in several applications such as manufacturing processes, medical measurements, activity trackers, autonomous vehicles, power consumption, etc. In a number of cases, the time series consists of high-frequency floating-point data. This data typically contains measurement noise due to which lossy compression can provide significantly better compression without adversely impacting the downstream applications. In some cases, lossy compression can lead to improvement in the performance of downstream applications due to implicit denoising of the data [21]. In several of these scenarios, the compression is performed on an edge device that collects the data and transmits it to the cloud. These edge devices are usually computation and communication bandwidth constrained and hence the compression solutions must be real time and suitable for

these devices.

A lossy compressor consists of an encoder and a decoder, where the encoder compresses the original time series $x_1, x_2, \ldots, x_n$ to obtain the compressed bit stream. This compressed bit stream can then be decompressed by the decoder to obtain the reconstructed time series $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_n$ which differs from the original time series to an extent acceptable for a specific application. The error in the reconstructed time series with respect to the original time series is measured in terms of a distortion function, such as the mean squared error, mean absolute error or maximum absolute error. The optimal distortion function is highly dependent on the downstream applications that work with the reconstructed time series, but in absence of specific domain knowledge, maximum absolute error, defined as $\max_{i=1,\ldots,n} |x_i - \hat{x}_i|$, is a generally acceptable distortion measure. Note that a maximum absolute error of ϵ implies that the original and the reconstructed time series differ by at most ϵ at any time step.

3.1.1 Our Contributions

We propose LFZip (Lossy Floating-point Zip), a lossy compressor for time series data based on the prediction-quantization-entropy coder framework [22] which achieves significant improvement in compression over the previous state-of-the-art compressors. LFZip works under the maximum absolute error distortion metric where the maximum allowable absolute error is a user-specified parameter. LFZip also supports compression of multivariate time series where the dependencies across the variables are exploited to further boost the compression.

LFZip is available as an open-source tool on GitHub, providing a easily extensible framework supporting several prediction models including linear predictors and neural networks. The GitHub URL is `https://github.com/shubhamchandak94/LFZip`.

3.1.2 Previous Work

There have been several works on lossless and lossy compression of floating-point time series and multidimensional scientific data. For lossless compression of floating-point data, specialized compressors such as FPZIP [23] outperform general-purpose

compressors on multidimensional datasets. Several lossy compressors have also been proposed, which allow much better compression at the expense of some acceptable level of distortion. We next discuss a few of the existing works in literature.

Swinging door [24] and Critical Aperture [25] algorithms retain only a subset of the points in the time series based on the maximum error constraint and use linear interpolation during decompression. These are widely used in certain domains [26, 27] due to their suitability for computationally constrained systems. Another line of work uses polynomial or regression models to predict the next point in the time series and then quantizes the prediction error. Compressors in this category include SZ [28, 29, 30, 31], ISABELA [32] and NUMARCK [33], of which SZ is the state-of-the-art compressor under maximum error distortion. Finally, some of the compressors [34, 35] apply a transform to the original time series and then perform quantization in the transformed domain. Note that some of these compressors are not specific to time series and also support multidimensional scientific datasets. We refer the reader to [28] for a more detailed survey on lossy compressors under different distortion measures.

We should note that the general approach of prediction-quantization-entropy coding employed in LFZip has been extensively applied for lossy compression in the context of speech [36], images [37] and videos [38], where domain-specific prediction models and distortion measures are used, and has also been studied theoretically (see [22]). While working within this traditional framework, LFZip attempts to utilize advances in prediction models to achieve improved error-bounded lossy compression of time series data. This follows a line of similar efforts to improve lossless data compression using powerful prediction models such as neural networks [39, 40, 41].

We next discuss the methods employed in the proposed compression framework.

3.2 Methods

3.2.1 Encoding and Decoding Framework

The encoding and decoding framework for LFZip is summarized in Figure 3.1. The encoder processes the input one symbol at a time, and consists of 3 stages: predictor, quantizer and entropy coder. For simplicity, we describe these in detail for the case of univariate time series. In case of multivariate time series, only the prediction step is modified while the quantizer and entropy coder act independently on each variable.

Predictor: At every time-step t, the predictor block tries to guess the value x_t, based on the past reconstructions. Formally, the predictor block fits a function $P_t(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_{t-1})$, to obtain the prediction y_t. Note that the predictor is restricted to be causal in nature and is applied on the reconstructed values (rather than the input) so that the same procedure can be applied during the decompression. The predictor function P_t itself can be adaptive but must satisfy the causality constraint and should be trained only on the reconstructed values. More details about the predictors used in the framework and the training procedure are presented in Section 3.2.2.

For multivariate time series with k variables per time step, the prediction of time step t for the j^{th} variable can be based on not only the time steps $1, \ldots, t-1$ for the j^{th} variable, but also on time steps $1, \ldots, t-1$ for the other variables and time step t for the first $j-1$ variables (to ensure causality). This allows the predictor to exploit dependencies across variables and provides improved compression in some cases.

Quantizer: The floating-point prediction error is quantized in this step so as to satisfy the maximum absolute error constraint. Let $\Delta_t = x_t - y_t$ be the difference between the true value and the predictor output at time step t. The quantizer performs uniform scalar quantization of this Δ_t using a step size of 2ϵ to obtain the quantized output $\hat{\Delta}_t$. The final reconstructed output $\hat{x}_t$ is then given by $\hat{x}_t = y_t + \hat{\Delta}_t$. Note that the uniform quantization guarantees that $|\hat{\Delta}_t - \Delta_t| \leq \epsilon$, which in turn implies that $|\hat{x}_t - x_t| \leq \epsilon$. We use 16-bit quantized output allowing for 65535 quantization bins (1 bin reserved for outliers as discussed below). We also tested 8-bit quantization,

but the impact was negligible since the entropy coding stage is able to remove any redundancy introduced by 16-bit quantization.

In practice, the presence of outliers or sudden change in statistics of the time series can lead to a large Δ_t value that lies outside the quantization range. In such cases, the reserved quantization bin is used and the data point x_t is stored as a floating-point number, with the reconstruction $\hat{x}_t = x_t$.

Entropy Coder: The final stage of the compression involves applying a universal lossless compressor on the quantized time series of the differences: $\hat{\Delta}_1, \hat{\Delta}_2, \ldots, \hat{\Delta}_n$ to obtain the variable length bit-stream b. LFZip uses BSC [42], an efficient BWT-based compressor, for this stage.

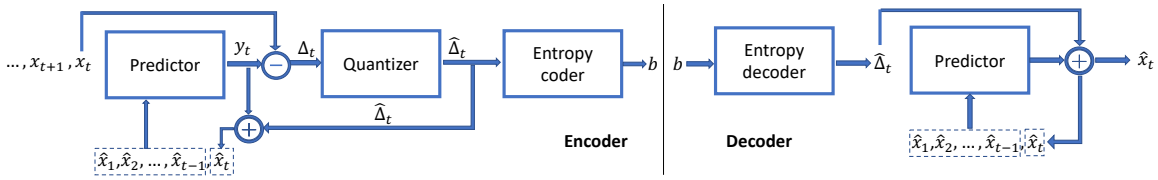


Figure 3.1: Encoder and decoder framework in LFZip.

The Decoder operations are symmetric but occur in the reverse order (see Figure 3.1). First the entropy-coded bit-stream b is decoded to obtain $\hat{\Delta}_1, \hat{\Delta}_2, \ldots, \hat{\Delta}_n$. The reconstruction then occurs one time-step at a time, in a causal fashion. At each time-step, the predictor $P_t(\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_{t-1})$ causally outputs the prediction y_t, which can then be used to output the reconstruction $\hat{x}_t$. Note that the same predictor function P_t and adaptive training procedure must be used during the encoding and decoding.

3.2.2 Predictors

Several prediction models can be utilized in the framework described. We look here at two classes of predictors of varying complexity supported by LFZip.

Normalized Least Mean Square Predictor (NLMS): The Normalized Least Mean Square Predictor (NLMS) can be thought of as an adaptive linear prediction filter of window size k (32 by default). The parameters of the linear filter are initialized with a fixed value and are updated at each time-step based on the mean square

prediction error. The update procedure is similar to stochastic gradient descent, where the gradients are normalized before update. As the predictor contains very few parameters, we observed that in practice it requires no pre-training and adapts very quickly to changing input statistics. In practice, the NLMS predictor works well on various types of inputs and is the default predictor in the current implementation.

Neural Network based Predictors: LFZip also supports more complex neural network based predictors. The current implementation supports different variants of the Fully Connected (FC) and the biGRU networks [43] for univariate time series. The FC and biGRU networks take as input the window of past k reconstruction symbols ($k = 32$ by default) and output a floating-point prediction for x_t. As the NLMS predictor can be thought of as a single layer linear neural network, the FC and biGRU networks are strictly stronger models (in terms of expressibility). However, the larger number of parameters in FC and biGRU networks make them adapt much more slowly to the changing statistics in the time series. To resolve this issue, we employ offline training for neural network based predictors before the encoding step.

During the offline training, the model is trained on some given training data with early stopping performed with respect to validation data. The trained model is used as the predictor during compression, and the parameters can be optionally updated online during the compression. The utility of offline training highly depends upon on the similarity of the training data with the test data being compressed.

Note that while the training is performed on the true unquantized values, during the compression, the model performs prediction based on the quantized values. This can lead to worse performance in the case where the maximum error threshold ϵ is large. To resolve this issue, we approximately emulate the quantization process during the training by adding some appropriate noise to the inputs (based on the maximum error parameter ϵ). We observed that adding noise during the training leads to 5-10% improvement in compression. We experimented with uniform and Gaussian noise models and observed that uniformly distributed noise model typically works better for the maximum error constraint.

3.3 Results

Name	Length	Description	BSC lossless compression ratio
acc	3.54M	Heterogeneity Activity Recognition - smartwatch accelerometer [44]	2.84
gyr	3.21M	Heterogeneity Activity Recognition - smartwatch gyroscope [44]	2.79
pow	2.05M	Household electric power consumption - active power [45]	5.21
ppg	0.50M	Blood volume pulse/photoplethysmography (PPG) [46]	2.48
gas	0.93M	Home activity monitoring - MOX gas sensors resistance [47]	4.97
dna	1.17M	Nanopore DNA sequencing raw current data	4.55
vib	1.55M	Siemens healthy tool vibration data	1.79
sen	0.75M	Siemens sensor data	4.27

Table 3.1: Datasets used for evaluation.

3.3.1 Experimental setup

We evaluated the proposed compressor LFZip on several time series datasets, spanning a variety of domains including smartwatch sensor data, household power consumption data, gas sensor array data, medical and genomic data, etc. as shown in Table 3.1. We also report results on two datasets obtained from Siemens. The first five datasets were chosen as a representative sample of the floating-point time series datasets from the UCI Machine Learning Repository [48]. Figure 3.2 contains snapshots of all the datasets. The datasets can be accessed online on the LFZip GitHub repository (except for the Siemens datasets).

We compare LFZip with two additional compressors SZ [28, 29, 30, 31] and critical aperture (CA) [25]. SZ is the current state-of-the-art compressor for maximum error distortion [28], while CA is widely used in the industry due to its low computational requirements [26, 27]. For SZ, we used the implementation available at `https://github.com/disheng222/SZ` (version: 2.1.7), while we implemented CA based on the description available at [25, 26, 49]. While certain implementations of CA do not use an entropy coding step, we apply BSC to the retained points for fair comparison with LFZip and SZ. Some of the datasets contained multivariate time series out of which a single variable was considered for fair comparison with CA and SZ as they

don't natively support multivariate time series compression.[1] Results for multivariate time series compression using LFZip are discussed in Section 3.3.3.

3.3.2 Results for LFZip (NLMS) for univariate time series data

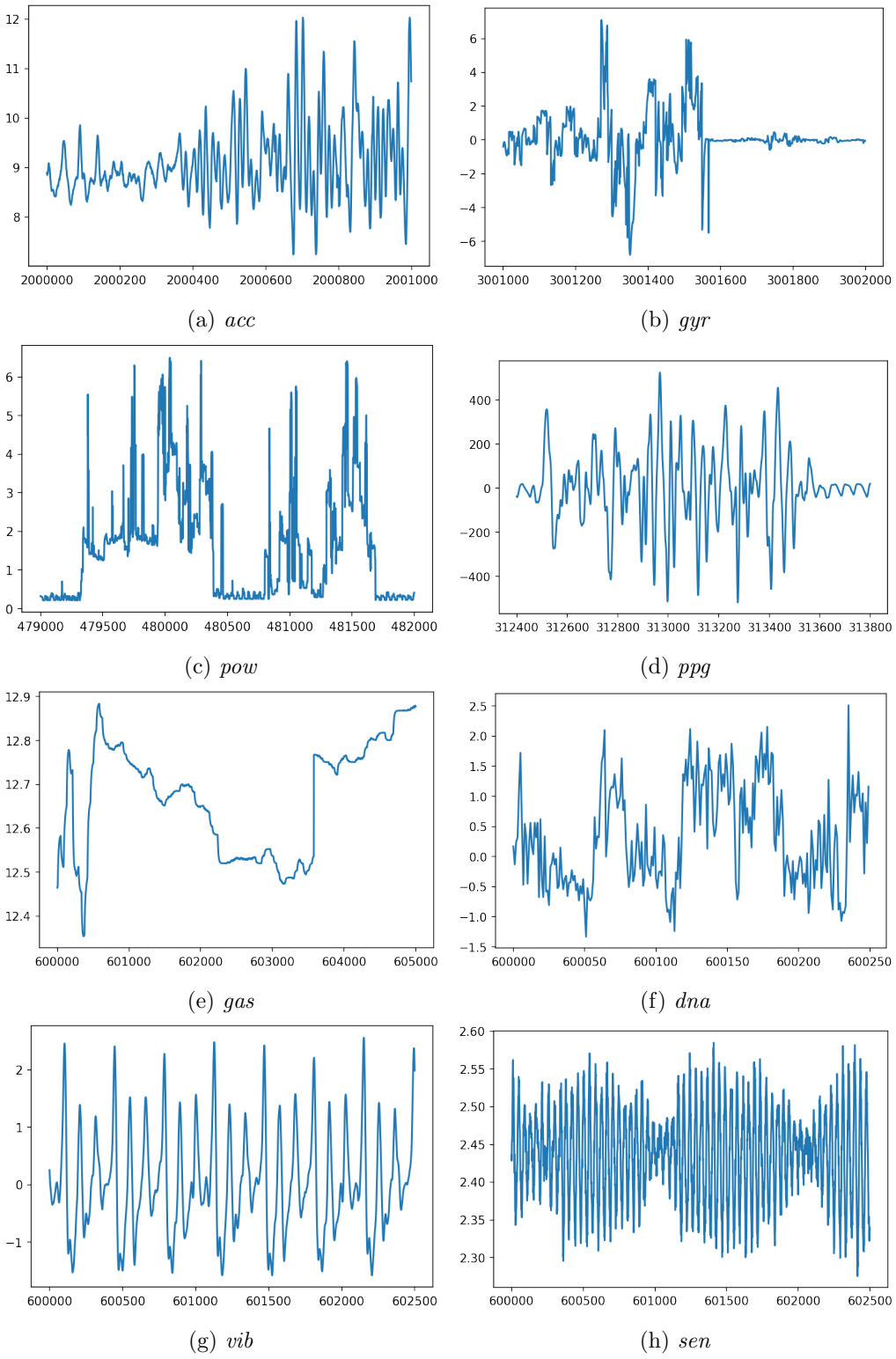
Dataset	Compressor	Maximum error ϵ		
		10^{-3}	10^{-2}	10^{-1}
acc	CA	2.84	3.01	5.19
	SZ	3.25	5.05	11.00
	LFZip (NLMS)	**3.55**	**5.86**	**12.71**
gyr	CA	2.88	4.27	10.75
	SZ	4.26	8.08	24.79
	LFZip (NLMS)	**6.05**	**12.26**	**28.77**
pow	CA	5.05	6.23	12.47
	SZ	**5.09**	**9.65**	**23.99**
	LFZip (NLMS)	4.17	7.37	17.98
ppg	CA	2.48	2.49	2.74
	SZ	2.43	2.80	4.39
	LFZip (NLMS)	**3.18**	**5.28**	**9.13**

Dataset	Compressor	Maximum error ϵ		
		10^{-3}	10^{-2}	10^{-1}
gas	CA	16.97	64.36	245.51
	SZ	22.69	75.84	**299.65**
	LFZip (NLMS)	**31.56**	**101.48**	252.55
dna	CA	**4.54**	4.54	4.86
	SZ	4.03	**4.55**	**8.62**
	LFZip (NLMS)	3.04	4.48	8.40
vib	CA	2.07	4.85	18.51
	SZ	4.77	11.77	40.61
	LFZip (NLMS)	**10.64**	**22.36**	**53.15**
sen	CA	4.34	7.60	125.04
	SZ	6.55	20.58	179.87
	LFZip (NLMS)	**6.88**	**21.70**	**180.98**

Table 3.2: Compression ratios for CA, SZ and LFZip (NLMS). Best results are boldfaced.

Table 3.2 shows the results for CA, SZ and LFZip (using NLMS predictor with default window size $k = 32$) for the eight univariate time series datasets and three values of the maximum error $(10^{-1}, 10^{-2}, 10^{-3})$. For comparison, Table 3.1 shows the results for lossless compression with BSC, where BSC was chosen since it outperformed other lossless tools like Gzip, 7-zip, bzip2 and fpzip [23]. We see that lossy compression can lead to significant benefits over lossless compression, especially at higher maximum error constraints. From Table 3.2, we see that LFZip achieves the best compression in most cases, except for the *pow* and *dna* datasets. In general we found that the LFZip (NLMS) offers the most benefits for datasets that are difficult to approximate with piecewise linear functions or lower order polynomials (see Figure 3.2 for snapshots of the datasets). The worse performance of LFZip (NLMS) on *pow*

[1]Note that although SZ supports multidimensional data compression, we found that this mode in fact leads to worse compression when run on multivariate time series, probably because SZ expects continuity along each dimension, which might not be true for the variables at a single time step.



(a) *acc* (b) *gyr*

(c) *pow* (d) *ppg*

(e) *gas* (f) *dna*

(g) *vib* (h) *sen*

Figure 3.2: Snapshots of the datasets used for evaluation.

and *dna* datasets can be explained by the sudden jumps in the time series which are difficult to predict using a linear model. We will see in Section 3.3.5 that we can overcome this limitation of NLMS predictors by using more powerful NN based predictors.

3.3.3 Results for LFZip (NLMS) for multivariate time series data

Dataset	Mode	Maximum error ϵ		
		10^{-3}	10^{-2}	10^{-1}
acc (X, Y, Z)	univariate	3.588	5.931	13.220
	multivariate	3.592	5.934	13.250
gyr (X, Y, Z)	univariate	6.295	13.605	34.181
	multivariate	6.409	13.763	34.597

Dataset	Mode	Maximum error ϵ		
		10^{-3}	10^{-2}	10^{-1}
gas (8 sensors)	univariate	26.239	63.304	152.378
	multivariate	27.614	75.179	204.006
sen (3 sensors)	univariate	6.627	19.669	166.568
	multivariate	6.669	20.334	304.878

Table 3.3: LFZip (NLMS) compression ratios for multivariate time series (i) when each variable is compressed independently and (ii) when compressed together.

LFZip can provide further improvement in compression for multivariate time series with dependencies across the variables. Table 3.3 shows the results for *acc* (3 variables: X, Y and Z), *gyr* (3 variables: X, Y and Z), *gas* (8 variables: different MOX gas sensors) and *sen* (3 variables: different sensors) where the different variables are (i) compressed independently with LFZip in the univariate mode and (ii) compressed together with LFZip in the multivariate mode. We see significant gains of compressing the series together for the *gas* and *sen* datasets, but very minor improvement for *acc* and *gyr* datasets. The advantages for compressing multivariate time series together is likely to be most pronounced when the values for other variables can aid the predictor beyond the past values for the same variable.

3.3.4 LFZip (NLMS) ablation experiments

In this section, we study the impact of the prediction and entropy coding stages for LFZip (NLMS). Figure 3.3 shows the impact of the NLMS window size (k) on the

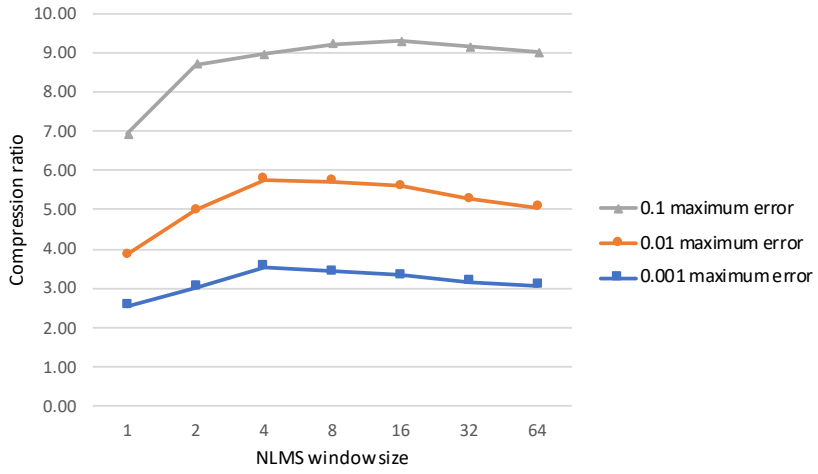


Figure 3.3: Compression ratio for *ppg* dataset as the NLMS window size is varied.

compression ratio for the *ppg* dataset. We observe that the performance initially improves with increasing k as the model becomes more expressive but at higher window sizes the performance becomes worse. This is likely because the higher number of parameters makes it slower at adapting to the changing statistics in the time series.

To understand the impact of the entropy coding stage on the performance of LFZip, we replaced the entropy coder BSC by 7-zip and Gzip for the *ppg* dataset with maximum error 10^{-2}. The compression ratios obtained were 5.28, 4.43 and 3.48 for BSC, 7-zip and Gzip, respectively. This shows that the entropy coder plays an important role in the framework and allows us to use a simple uniform scalar quantizer. We note that even with 7-zip as the entropy coder, LFZip achieves better compression than SZ for most datasets, showing the advantages of improved prediction.

Finally, we note that the change in compression ratio as the NLMS window size changes depends on the specific dataset. For certain datasets, such as the *dna* dataset, we found that a window size of 0 performed the best. Note that a window size of 0 corresponds to removing the predictor and directly quantizing the time series and compressing the quantized values with BSC. We believe that there is an interesting interaction between the predictor attempting to minimize the loss for the error sequence, and the BSC entropy coder directly being able to capture the redundancy in the quantized time series. Further exploration of this phenomenon is part of future

work.

3.3.5 Results for LFZip (NN) for univariate time series data

Dataset	Compressor	Maximum error ϵ 10^{-2}	10^{-1}
acc	SZ	4.64	9.38
	LFZip (NLMS)	5.10	10.19
	LFZip (NN)	**5.26**	**10.78**
gyr	SZ	6.99	20.96
	LFZip (NLMS)	10.22	23.33
	LFZip (NN)	**10.35**	**25.00**

Dataset	Compressor	Maximum error ϵ 10^{-2}	10^{-1}
pow	SZ	**9.44**	23.57
	LFZip (NLMS)	7.21	17.74
	LFZip (NN)	9.29	**25.38**
dna	SZ	4.45	8.67
	LFZip (NLMS)	4.46	8.40
	LFZip (NN)	**4.60**	**8.99**

Table 3.4: Compression ratios for test datasets for SZ, LFZip (NLMS) and LFZip (NN). Best results are boldfaced.

Table 3.4 shows the results for LFZip with a neural network (NN) based predictor. For these experiments, we used a simple fully connected network with 4 hidden layers with 128 neurons each and first layer with 32 neurons, ReLU activation and batch normalization. Only the datasets for which the statistics were stationary over time were used for these experiments so as to allow offline training before compression. The datasets were divided into training, validation and test datasets of equal sizes, and the compression was performed only on the test dataset for all the compressors. For LFZip (NN), offline training was performed using the training and validation datasets for 5 epochs. During training, uniformly distributed noise in $[-0.05, 0.05]$ was added to emulate the quantization noise. No adaptive training was employed during compression as that didn't seem to provide measurable benefits.

From Table 3.4, we see that LFZip (NN) improves the result over LFZip (NLMS) and SZ in all cases, except for *pow* dataset with maximum error 10^{-2} where it is slightly worse than SZ. This shows that better prediction using stronger NN based models can lead to further improvement in compression for LFZip at the cost of increased computational complexity. We also observed that in certain cases, removing the predictor altogether outperforms both NN and NLMS, suggesting that our understanding of the underlying dynamics between the predictor and the entropy coder remains incomplete.

3.3.6 Computational requirements

The experiments were run on an Ubuntu 18.04 server with 2.2 GHz Intel Xeon processors and NVIDIA TITAN X Pascal GPUs. Note that the GPUs were used only during the training phase of the NNs, and the compression with both NLMS and NN predictors was performed on CPU with a single thread to ensure reproducibility and correct decompression. The current implementation of LFZip (NLMS) is written in Python and C++, and achieved a compression/decompression speed of ∼2M timesteps/s for univariate time series. This is about an order of magnitude slower than SZ but should be practical for most applications. LFZip (NN) is more computationally expensive with a speed of ∼1K timesteps/s for the model used above.

3.4 Conclusion

In this chapter, we proposed an error-bounded lossy compressor for multivariate floating-point time series based on the prediction-quantization-entropy coder framework. Using linear and neural network prediction models, the proposed compressor LFZip achieves higher compression ratios than the previous state-of-the-art compressors. Future work includes an optimized implementation for the neural network-based framework, extension of the framework to multidimensional datasets, and exploration of other predictive models to further boost compression. Another direction to explore is the interplay between the predictor and the entropy coder, and possible joint optimization of the two in order to maximize the compression ratios.

Chapter 4

Impact of lossy compression of nanopore raw signal data on basecalling and consensus accuracy

4.1 Introduction

Nanopore sequencing technologies developed over the past decade provide a real-time and portable sequencing platform capable of producing long reads, with important applications in completing genome assemblies and discovering structural variants associated with several diseases [50]. Nanopore sequencing consists of a membrane with pores where DNA passes through the pore leading to variations in current passing through the pore. This electrical current signal is sampled to generate the raw signal data for the nanopore sequencer and is then basecalled to produce the read sequence. Due to the continuous nature of the raw signal and high sampling rate, the raw signal data requires large amounts of space for storage, e.g., a typical 30x depth human sequencing experiment can produce terabytes of raw signal data, which is an order of magnitude more than the space required for storing the basecalled reads [51].

Due to the ongoing research into improving basecalling technologies and the scope for further improvement in accuracy, the raw data needs to be retained to allow repeated analysis of the sequencing data. This makes compression of the raw signal

data crucial for efficient storage and transport. There have been a couple of lossless compressors designed for nanopore raw signal data, namely, Picopore [52] and VBZ (`https://github.com/nanoporetech/vbz_compression/`). Picopore simply applies gzip compression to the raw data, while VBZ, which is the current state-of-the-art tool, uses variable byte integer encoding followed by zstd compression. Although VBZ reduces the size of the raw data by 60%, the compressed size is still quite significant and further reduction is desirable. However, obtaining further improvements in lossless compression is challenging due to the inherently noisy nature of the current measurements.

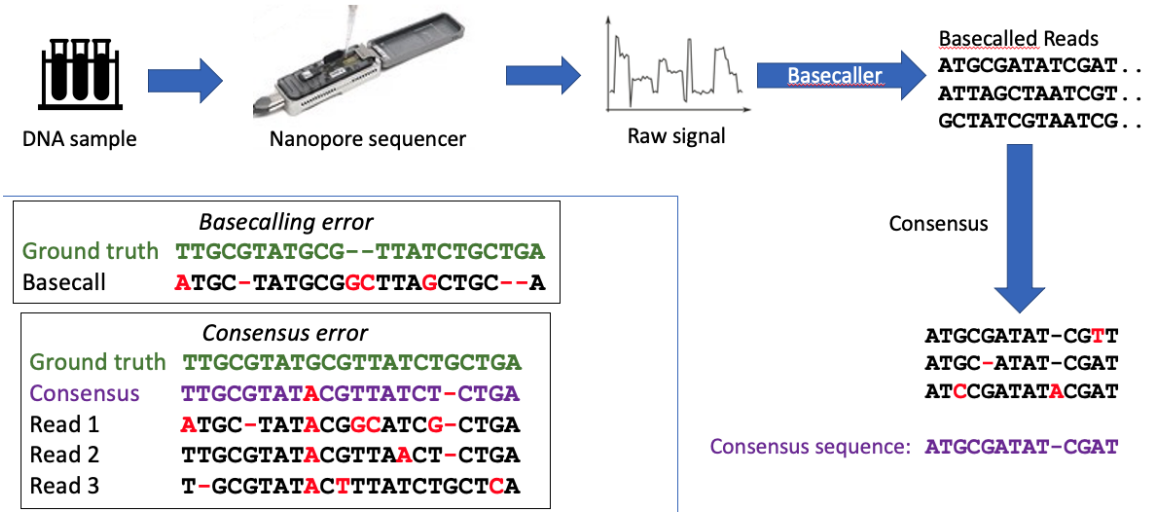


Figure 4.1: Illustration of nanopore sequencing, basecalling and consensus. The bottom-left panel shows instances of basecalling and consensus errors, where the consensus process is able to correct random errors but not systematic errors.

In this context, lossy compression is a natural candidate to provide a boost in compression at the cost of certain amount of distortion in the raw signal. There have been several works on lossy compression for time series data, including SZ [53] and LFZip (Chapter 3) that provide a guarantee that the reconstruction lies within a certain user-defined interval of the original value for all time steps. However, in the case of nanopore raw current signal, the metric of interest is not the maximum deviation from the original value, but instead the impact on the performance of basecalling and

other downstream analysis steps. In particular, two quantities of interest are the basecalling accuracy and the consensus accuracy. The basecalling accuracy measures the similarity of the basecalled read sequence to the known true genome sequence, while the consensus accuracy measures the similarity of the consensus sequence obtained from multiple overlapping reads to the known true genome sequence. As discussed in [54], these two measures are generally correlated but can follow different trends in the presence of systematic errors. In general, consensus accuracy can be thought of as the higher-level metric, which is usually of interest in most applications, while basecalling accuracy is a lower-level metric in the sequencing analysis pipeline. Figure 4.1 illustrates the sequencing, basecalling and consensus process.

In this chapter, we study the impact of lossy compression of nanopore raw signal data on basecalling and consensus accuracy. We evaluate the results for several basecallers and at multiple stages of the consensus pipeline to ensure the results are generalizable to future iterations of these tools. We find that lossy compression using general-purpose tools can provide significant reduction in file sizes with negligible impact on accuracy. To further stress-test the ability of lossy compression to preserve useful information, we look into the impact of lossy compression on methylation calling performance and reach similar conclusions as those for basecalling accuracy. To the best of our knowledge, this is the first study exploring the use of lossy compression for nanopore raw signal data and performing a systematic analysis of its impact on downstream applications. We believe our results provide motivation for research into specialized lossy compressors for nanopore raw signal data and suggest the possibility of reducing the resolution of the raw signal generated on the nanopore device itself while preserving the downstream performance. The source code and data for our analysis is publicly available at `https://github.com/shubhamchandak94/lossy_compression_evaluation` and can be useful as a benchmarking pipeline for further research into lossy compression for nanopore data.

4.2 Background

4.2.1 Nanopore sequencing and basecalling

Nanopore sequencing, specifically the MinION sequencer developed by Oxford Nanopore Technologies (ONT) [50], involves a strand of DNA passing through a pore in a membrane with a potential applied across it. Depending on the sequence of bases present in the pore (roughly 6 bases at any instant), the ionic current passing through the pore varies with time and is measured at a sampling frequency of 4 kHz. The sequencing produces 5-15 current samples per base, which are quantized to a 16-bit integer and stored as an array in a version of the HDF5 format called fast5. The current signal is then processed by the basecaller to generate the basecalled read sequence and the corresponding quality value information. In the uncompressed format, the raw current signal requires close to 18 bytes per sequenced base which is significantly more than the amount needed for storing the sequenced base and the associated quality value. The sequenced FASTQ files can also be compressed further using specialized compressors [55] to further reduce the storage costs.

Over the past years, there has been a shift in the basecalling strategy from a physical model-based approach to a machine learning-based approach leading to significant improvement in basecalling accuracy (see [54] and [56] for a detailed review). In particular, the current default basecaller Guppy by ONT (based on open source tool Flappie (`https://github.com/nanoporetech/flappie`)) uses a recurrent neural network that generates transition probabilities for the bases which are then converted to the most probable sequence of bases using Viterbi algorithm. Another recent basecaller by ONT is bonito (`https://github.com/nanoporetech/bonito/`, currently experimental), which is based on a convolutional neural network and CTC decoding [57], achieving close to 92-95% basecalling accuracy in terms of edit distance. Despite the progress in basecalling, the current error rates are still relatively high (typically 5-10%) with considerable fraction of insertion and deletion errors, which necessitates the storage of the raw data for utilizing improvements in the basecalling technologies for future (re)analysis.

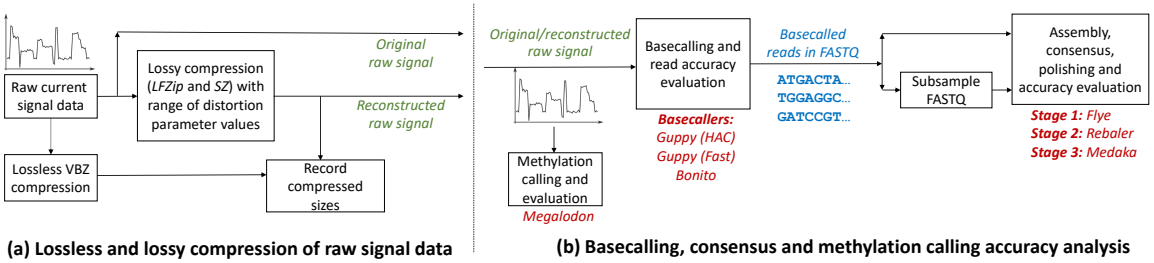


Figure 4.2: Flowchart showing the experimental procedure. (a) The raw data was compressed with both lossless and lossy compression tools, (b) the original and lossily compressed data was then basecalled with three basecalling tools. Finally, the basecalled data and its subsampled versions were assembled and the assembly (consensus) was polished using a three-step pipeline (1. Flye, 2. Rebaler, 3. Medaka). The tradeoff between compressed size and basecalling/consensus accuracy was studied. For one dataset, methylation calling and accuracy evaluation was performed using Megalodon.

4.2.2 Assembly, consensus and polishing

Long nanopore reads allow much better repeat resolution and are able to capture long-range information about the genome leading to significant improvements in de novo genome assembly [51]. However genome assembly with nanopore data needs to handle the much higher error rates as compared to second generation technologies such as Illumina, and there have been several specialized assemblers for this purpose, including Flye [58, 59] and Canu [60], some of which allow hybrid assembly with a combination of short and long read data.

Nanopore de novo assembly is usually followed by a consensus step that improves the assembly quality by aligning the reads to a draft assembly and then performing consensus from overlapping reads (e.g., Racon [61]). Note that the consensus step can be performed even without de novo assembly if a reference sequence for the species is already available, in which case the alignment to the reference is used to determine the overlap between reads. Further polishing of the consensus sequence can be performed with tools specialized for nanopore sequencing that use the noise characteristics of the sequencing and/or basecalling to find the most probable consensus sequence. For example, Nanopolish [62] directly uses the raw signal data for polishing the consensus

using a probabilistic model for the generation of the raw signal from the genomic sequence. Medaka (`https://nanoporetech.github.io/medaka/`) is the current state-of-the-art consensus polishing tool both in terms of runtime and accuracy (`https://github.com/rrwick/August-2019-consensus-accuracy-update/`). Medaka uses a neural network to perform the consensus from the pileup of the basecalled reads at each position of the genome.

4.2.3 Methylation calling

DNA methylation plays an important role in various biological functions [63], with 6mA and 5mC being the most commonly studied methylated bases (methylated versions of A and C, respectively). Since nanopore sequencing can work with native (non-PCR amplified) DNA, it is possible to detect methylated bases due to the changes in the raw signal when the methylated base passes through the pore. This fact has been exploited to develop methylation calling pipelines using various techniques including hidden Markov model (HMM)-based methods and neural network-based methods [62, 63, 64, 65]. Here, we use Megalodon (`https://github.com/nanoporetech/megalodon/`) which first anchors the intermediate probabilities produced by the basecalling neural network (from Guppy basecalling modes that call both modified and unmodified bases) to the reference sequence. It then uses traditional HMM algorithms such as Viterbi and forward-backward algorithms [66] to compute the probability that a given base is modified. This common framework can be used to call different types of base modifications by providing the appropriate basecalling model, and we focus on CpG motifs on the human genome in this chapter.

4.2.4 Lossy compression

Lossy compression [67] refers to compression of the data into a compressed bitstream where the decompressed (reconstructed) data need not be exactly but only approximately similar to the original data. Lossy compression is usually studied in the context of a distortion metric that specifies how the distortion or error between the original

and the reconstruction is measured. This gives rise to a tradeoff between the compressed size and the distortion, referred to as the rate-distortion curve. Here, we work with two state-of-the-art lossy compressors for time-series data, LFZip (Chapter 3) and SZ [53]. Both these compressors work with a maxerror parameter that specifies the maximum absolute deviation between the original and the reconstructed data. If $x_1, \ldots, x_T$ denotes the original time-series, $\hat{x}_1, \ldots, \hat{x}_T$ denotes the reconstructed time-series, and ϵ is the maxerror parameter, then these compressors guarantee that $\max_{t=1,\ldots,T} |x_t - \hat{x}_t| < \epsilon$.

LFZip and SZ use slightly different approaches towards lossy compression. LFZip uses a prediction-quantization-entropy coding approach, and here we use a mode wherein the prediction step is skipped and LFZip simply performs uniform scalar quantization (i.e., uniform binning with the bin size determined by the maxerror) followed by entropy coding. This mode provided the best compression in our experiments. SZ uses a curve fitting step followed by entropy coding, with the reconstruction lying on a low-degree local polynomial approximation to the original data.

There are a couple of reasons for focusing on lossy compressors with maximum absolute deviation as the distortion metric instead of mean square error or mean absolute error in this work. The first reason is that the guaranteed maximum error implies that the reconstructed raw signal is close to the original value at each and every timestep and not only in the average sense. Hence, the maximum error distortion metric is preferable for general applications where the true distortion metric is not well understood. The second reason is the availability of efficient implementations which is crucial for compressing the large genomic datasets. However, we believe that there is significant scope for using mean square error and other metrics for developing specialized lossy compressors for nanopore data given the better theoretical understanding of those metrics.

4.3 Experiments

We next describe the experimental setup in detail (see Figure 4.2 for a flowchart representation). The instructions for downloading the datasets and installing the

tools, as well as the scripts for performing the experiments are available on the GitHub repository. The experiments were run on an Ubuntu 18.04.4 server with 40 Intel Xeon processors (2.2 GHz), 260 GB RAM and 8 Nvidia TITAN X (Pascal) GPUs.

4.3.1 Datasets

Species	Sample	Genome size (bp)	GC-content	Flowcell type	Read count	Read length N50 (bp) [54]	Approx. depth	Raw signal size (GB)		Source
								Uncompressed	VBZ (lossless)	
Staphylococcus aureus	CAS38_02	2.9×10^6	32.8%	R9.4.1	11,047	24,666	83x	4.86	2.02	[54]
Klebsiella pneumoniae	INF032	5.1×10^6	57.6%	R9.4	15,154	37,181	108x	10.14	4.32	[54]
Escherichia coli	K-12 MG1655	4.6×10^6	50.8%	R10.3	92,000	7,431	128x	12.09	5.14	See Caption
Homo sapiens	NA12878	3.1×10^9	40.9%	R9.4	128,314	11,404	0.29x	25.37	10.31	[51]

Table 4.1: Datasets used for analysis. The *E. coli* dataset was obtained from `http://albertsenlab.org/we-ar10-3-pretty-close-now/`. N50 is a statistical measure of average length of the reads. The uncompressed size column refers to storing the raw signal in the default representation using 16 bits/signal value. The first three datasets (bacterial) were used for basecalling and consensus accuracy evaluation, while the last dataset (low-coverage human dataset from a single flowcell) was used for basecalling and per-read methylation calling accuracy evaluation.

Table 4.1 shows the datasets used for analysis in this work. The first three bacterial datasets were chosen to be representative datasets with different GC-content and flowcell types, including the latest R10.3 pore. We note that some of the tools were not run on the *E. coli* dataset since they do not yet support the R10.3 pore. For all these datasets the ground-truth genomic sequence is known through hybrid assembly with long and short read technologies. The table also shows the uncompressed and VBZ compressed sizes for the datasets, we observe that lossless compression can provide size reduction of roughly 60%. For each dataset, we run our analysis on both the original read depth as well as subsampled versions of the datasets (2x, 4x and 8x subsampling of Fastq files performed using Seqtk (`https://github.com/lh3/seqtk/`)). This helps us understand how the impact of lossy compression on consensus accuracy depends on read depth. The last dataset (human) is used for basecalling and methylation calling accuracy evaluation, and we use one flowcell (FAB45280) of NA12878 human nanopore data from [51] consisting of around 900M sequenced bases. For basecalling accuracy evaluation, we generate the ground truth genome by applying the variants

from GIAB [68] to the reference genome. More details on the methylation calling experiments are provided in Section 4.3.5.

We focus on using bacterial datasets rather than human datasets for consensus accuracy evaluation for a few reasons similar to those cited in [54]. Firstly, bacterial datasets typically have a more reliable ground truth allowing for more precise estimation of the impact of lossy compression. This is especially important for consensus accuracy which can be very high ($\gtrsim 99.9\%$), leading to a much greater uncertainty in the evaluation due to errors in the ground truth sequence. The smaller size for bacterial datasets also allows more extensive experimentation at higher coverage and across several parameters. Due to these reasons, previous studies have often relied on bacterial datasets for consensus/assembly accuracy evaluation, and used human datasets only for basecalling accuracy evaluation. (e.g., see the works [69] and [70] on novel basecalling algorithms). In addition, lossy compression with maximum deviation constraint is typically local in nature, with LFZip in particular performing uniform scalar quantization independently at each time step. Thus, the size of the genome should not impact the analysis and the results should generalize to larger genomes. Further experimentation on larger eukaryotic datasets remains part of future work as better benchmark datasets are obtained.

We also looked into the possibility of using data from Zymo microbial community standard [71] which has been used to evaluate basecallers and consensus tools. However, we decided to use the previously described datasets for a couple of reasons. Firstly, the Zymo dataset is a metagenomic dataset and obtaining data for individual species requires additional analysis and introduces a possibility of erroneous conclusions. Secondly, several neural network models in the downstream pipeline (e.g., basecallers and Medaka consensus) are commonly trained on parts of the Zymo dataset, with different tools using different training and testing genomes. This makes the data unsuitable when comparing several tools due to overfitting concerns.

4.3.2 Lossy compression

To study the impact of lossy compression, we generate new fast5 datasets by replacing the raw signal in the original fast5 files with the reconstruction produced by lossy compression. We use open source general-purpose time-series compressors LFZip (version 1.1) and SZ [53] (version 2.1.8.3). Both the tools require a parameter representing the maximum absolute deviation (maxerror) of the reconstruction from the original. We conducted ten experiments for each tool by setting the maxerror parameter to $1, 2, \ldots, 10$. To put this in context, note that for a typical current range of 60 pA and the typical noise value of 1-2 pA, the maxerror settings of 1 and 10 correspond to current values of 0.17 pA and 1.7 pA, respectively (`https://github.com/nanoporetech/kmer_models/`). Finally, we note that both LFZip and SZ can compress millions of timesteps per second and hence can be used to compress the nanopore raw signal data in real time as it is produced by the sequencer.

4.3.3 Basecalling and consensus

We perform basecalling on the raw signal data (both original and lossily compressed) using two modes of Guppy (version 3.6.1) as well as with bonito (version 0.2.0, note that bonito is currently an experimental release). For Guppy, we use the default high accuracy mode (guppy_hac) and the fast mode (guppy_fast). Both the modes use the same general framework but differ in terms of the neural network architecture size and weights. We use these three basecaller settings to study whether lossy compression leads to loss of useful information that can be potentially exploited by future basecallers.

We use a three-step assembly, consensus and polishing pipeline based on the analysis and recommendations in the addendum to [54]. The first step is de novo assembly using Flye (version 2.7.1) [58, 59] which produces a basic draft assembly. The second step is consensus polishing of the Flye assembly using Rebaler (`https://github.com/rrwick/Rebaler/`, version v0.2.0) which runs multiple rounds of Racon (version 1.4.13) to produce a high quality consensus of the reads. Finally,

the third step uses Medaka (version 1.0.3) by ONT that performs further polishing of the Rebaler consensus using a neural network-based approach. Note that the neural network model for Medaka needs to be chosen corresponding to the basecaller.

4.3.4 Evaluation metrics

For evaluating the basecalling and consensus accuracy, we use the pipeline presented for the task of basecaller comparison in [54] and its addendum (`https://github.com/rrwick/August-2019-consensus-accuracy-update/`). The basecalled reads were aligned to the true genome sequence using minimap2 [72] and the read's basecalled identity was defined as the number of matching bases in the alignment divided by the total alignment length. We report only the median identity across reads in the results section (see GitHub for details on accessing the per-read results). The consensus accuracy after each stage is computed in a similar manner, where instead of aligning the reads, we split the assembly into 10 kbp pieces and then find median identity across these pieces. Finally, we compute the basecalling and consensus Qscore using the Phred scale as $\text{Qscore} = -10\log_{10}(1 - \text{identity})$ where the identity is represented as a fraction. We refer the reader to [54] for further discussion on these metrics. In addition, we evaluate the accuracy of homopolymer sequences in the consensus using the `fastmer.py` script (`https://github.com/jts/assembly_accuracy`), since homopolymer calling has been identified as one of the main challenges of nanopore sequencing [56].

4.3.5 Methylation calling and evaluation

We consider the impact of lossy compression on methylation calling to understand whether the loss in information due to compression leads to further degradation of methylation calling performance as compared to basecalling, given that methylation calling is a more sensitive task than basecalling. For evaluating this impact, we use the pipeline and benchmark dataset used in two previous works, DeepMod [64] and DeepSignal [65]. We use NA12878 human nanopore data from [51] which used native (non-PCR amplified) DNA and use a benchmark obtained from bisulfite sequencing

from the ENCODE project (ENCFF835NTC) [73]. Following the procedure in [65], we identify the high confidence positive and negative sites on the genome by restricting ourselves to sites with coverage at least five, and 100% positive or negative calls on both strands in the bisulfite dataset. This resulted in roughly 5.4M positive and 4.7M negative high confidence sites on the genome.

We then used Megalodon (version 2.1.0) on the nanopore dataset to obtain a list of per-read methylation calls (predicted probabilities) for each CpG motif in the read. We used a basecalling model specially trained for calling CpG methylation released in the Rerio repository (`https://github.com/nanoporetech/rerio/`). We then computed the precision, recall and AUC (area under ROC curve), restricting ourselves to the high confidence sites determined above. For the precision and recall computation, we used a threshold of 0.5 for the predicted methylation probabilities.

As shown in Table 4.1, we used only one flowcell of data consisting of around 900M sequenced bases. For this study, we focused exclusively on per-read evaluation and did not attempt to compute correlation of the methylation frequencies with the bisulfite data as done in [65]. We believe that the per-read evaluation should be indicative of the extent to which lossy compression leads to loss of information regarding methylation. We also found other datasets in the previous works [63, 64, 65] where ground truth positive and negative datasets were generated using methyltransferase enzyme and PCR-amplification, respectively. However, these datasets were generated with older pores (R7 or 2D technology) which are not supported by the modern methylation calling tools. Better benchmark datasets in the future can be helpful for more extensive evaluation.

4.4 Results and discussion

We now discuss the main results obtained from the experiments described above. Throughout the results and discussion, the compressed sizes for lossy compression are shown relative to the compressed size for VBZ lossless compression, where the lossless compression sizes are shown in Table 4.1. Additional results and plots are available on GitHub.

Figure 4.3 shows the variation of the size of the lossily compressed dataset with the maxerror parameter. We see that LFZip generally provides better compression than SZ at the same maxerror value, although this fact by itself does not guarantee a better tradeoff for the metrics of interest. We also see that lossy compression can provide significant size reduction over lossless compression even with relatively small maxerror (recall that maxerror of 1 in the 16-bit representation of the raw signal corresponds to 0.17 pA error in the current value). For example, at maxerror of 5, lossy compression can provide size reduction of around 50% over lossless compression and size reduction of around 70% over the uncompressed 16-bit representation.

Both SZ and LFZip can compress millions of samples per second, with SZ being about an order of magnitude faster than LFZip (Chapter 3). Since LFZip simply performs uniform scalar quantization and entropy coding (in the mode used here), we believe that it can be significantly optimized further for this application. We also looked into the possibility that lossy compression could impact the speed/peak memory usage of the steps in the downstream pipeline (basecalling, assembly, etc.), however we observed that such effects were relatively small ($\lesssim$5-10% change) and mostly attributable to experimental variation. The time and memory usage results for various stages in the pipeline and the impact of lossy compression on these are provided in section 4.4.4.

4.4.1 Basecalling accuracy

Figure 4.4 shows the tradeoff achieved between basecalling accuracy and compressed size for the four datasets for all three basecallers. First, we observe that guppy_hac performs the best closely followed by bonito, and guppy_fast is typically far behind. As the maxerror parameter is increased, the basecalling accuracy stays stable for all the basecallers till the compressed size reaches 65% of the losslessly compressed size. For example, the basecalling accuracy for the *S. aureus* dataset with guppy_hac is $\sim$96.1% for lossless compression, and $\sim$96.0% at a 35% size reduction. After this the basecalling accuracy drops more sharply, becoming 2% lower than the original lossless level when the maxerror parameter is 10. The drop seems to follow a similar trend for

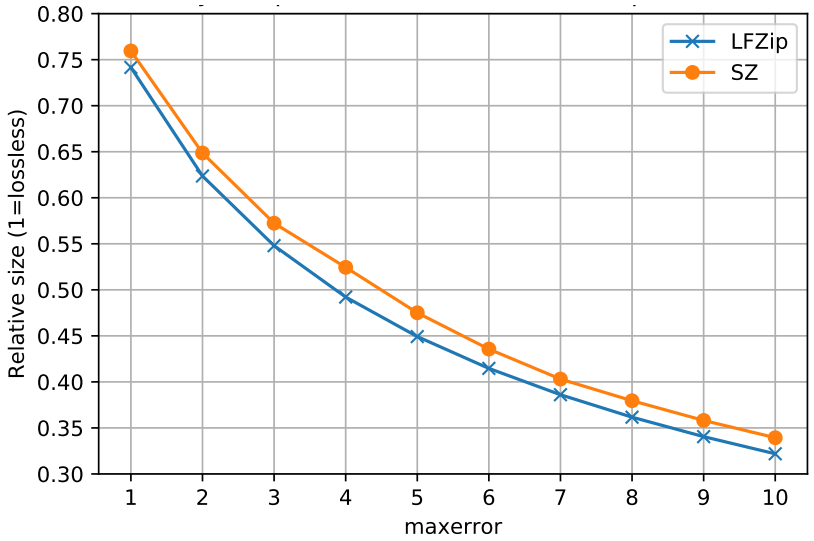


Figure 4.3: Compressed size for lossy compression with LFZip and SZ for the *S. aureus* dataset as a function of the maxerror parameter. The compressed sizes are shown relative to the VBZ lossless compression size.

all the basecallers and compressors, suggesting that at least 35% reduction in size over lossless compression can be obtained without sacrificing basecalling accuracy. Note that for maxerror parameter equal to 10, the allowed deviation of the reconstructed raw signal is larger than the typical noise levels in sequencing, and hence lossy compression probably leads to perceptible loss in the useful information contained in the raw signal.

We observed that the impact of lossy compression on read lengths and the number of aligned reads is negligible. This suggests that lossy compression generally leads to local and small perturbations in the basecalled read and does not lead to major structural changes in the read such as loss of information due to trimmed/shortened reads. This is expected given that the lossy compressors used here guarantee that the reconstructed signal is within a certain deviation from the original signal at each time step.

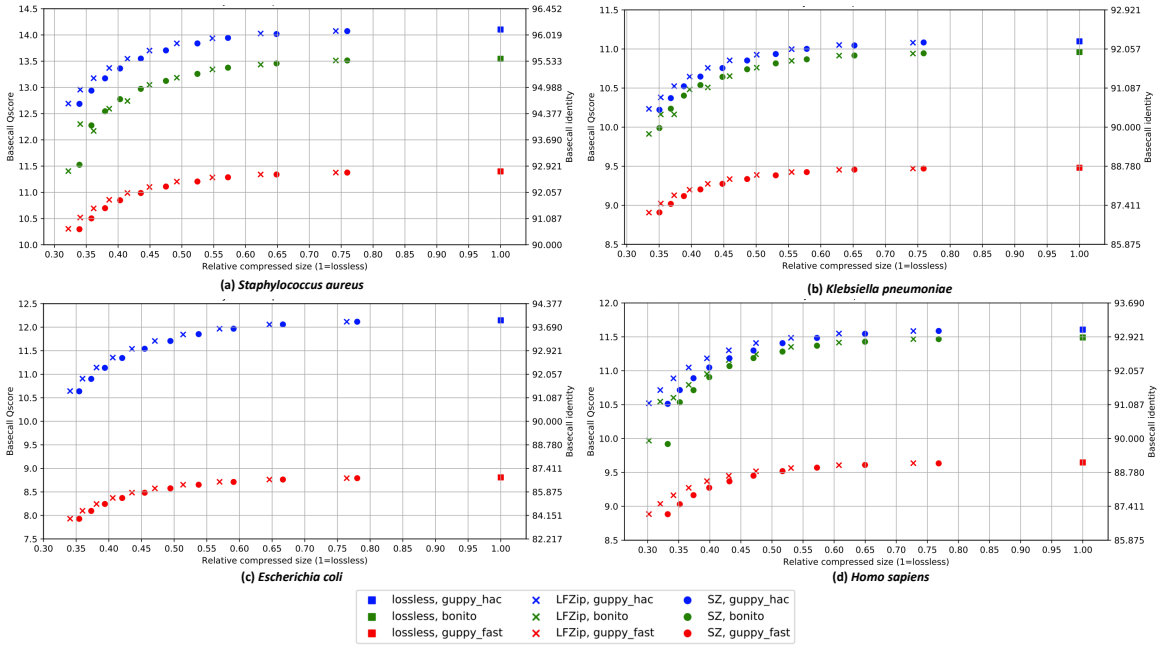


Figure 4.4: Basecalling accuracy vs. compressed size for (a) *S. aureus*, (b) *K. pneumoniae*, (c) *E. coli*, and (d) *H. sapiens* datasets. The results are displayed for the losslessly compressed data and the lossily compressed versions with LFZip and SZ (with maxerror 1 to 10) for the four basecallers. The compressed sizes are shown relative to the VBZ lossless compression size. Bonito was not run on *E. coli* due to lack of support for the R10.3 pore.

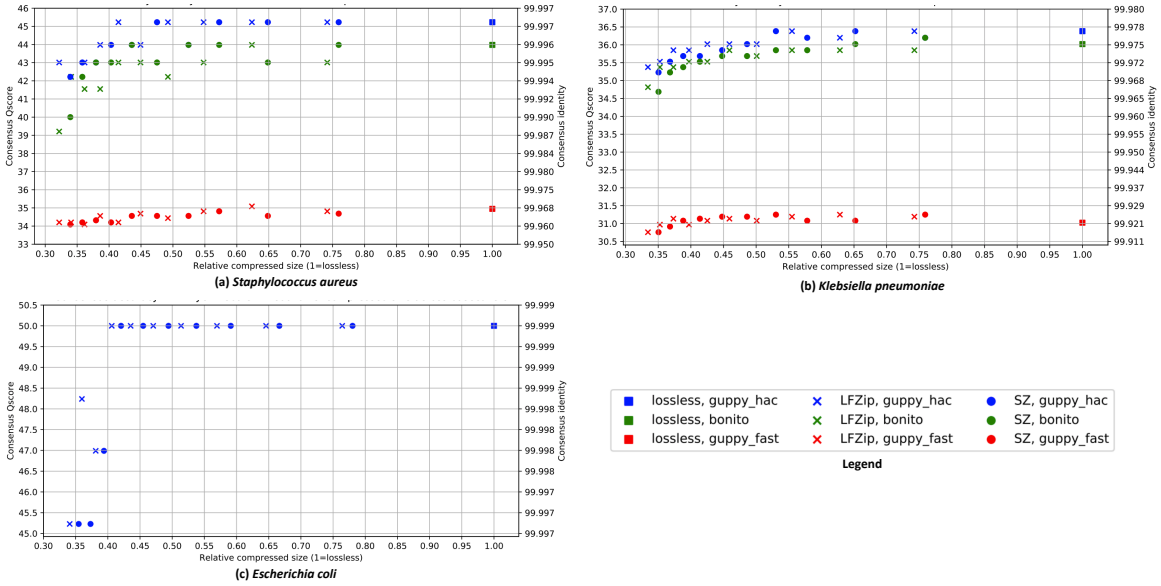


Figure 4.5: Consensus accuracy vs. compressed size for (a) *S. aureus*, (b) *K. pneumoniae* and (c) *E. coli* datasets. The results are displayed for the polished Medaka assembly for the losslessly compressed data and the lossily compressed versions with LFZip and SZ (with maxerror 1 to 10) for the four basecallers. The compressed sizes are shown relative to the VBZ lossless compression size. Bonito and guppy_fast were not used on *E. coli* due to lack of corresponding Medaka models for the R10.3 pore.

4.4.2 Consensus accuracy

Figures 4.5, 4.6(a) and 4.6(b) study the tradeoff between consensus accuracy and compressed size (i) across basecallers for the final Medaka polished assembly, (ii) across the assembly stages and (iii) across read depths in subsampled datasets, respectively. As expected, we observe that the consensus accuracy is significantly higher than basecalling accuracy across these experiments. We also observe that the consensus accuracy stays at the original lossless level till the compressed size reaches around 40-50% of the losslessly compressed size (50-60% reduction) and the drop in accuracy beyond this is relatively small. For example, the consensus accuracy for the *S. aureus* dataset with guppy_hac is ∼99.997% for lossless compression, and stays the same at a 50% size reduction. Thus, the impact of lossy compression on consensus accuracy is less severe than that on basecalling accuracy. This suggests that the errors introduced by lossy compression are generally random in nature and are mostly corrected by the consensus process.

In our experiments, lossy compression did not affect the number of assembled contigs (always 1) and the contig length in most cases. The only exceptions were the 4x and 8x subsampled versions of the *E. coli* dataset where the assemblies for both the lossless and the lossily compressed datasets were fragmented. This might be due to lower data quality as evidenced by the significantly smaller read lengths for this dataset (see Table 4.1). In general, this again suggests that the impact of lossy compression is localized and without large-scale disruptions in the assembly/consensus process, although further experiments on larger eukaryotic genomes might be required to strengthen this claim.

Figure 4.6(a) considers the consensus accuracy after each stage of the assembly/consensus process (Flye, Rebaler, Medaka) for the *S. aureus* dataset basecalled with guppy_hac. We see that each stage leads to further improvement in the consensus accuracy. We also observe that the earlier stages of the pipeline are impacted more heavily by lossy compression (in terms of percentage reduction in accuracy) than the final Medaka stage. This is expected since each successive stage of the assembly/consensus pipeline provides further correction of the basecalling errors caused due to lossy compression. This effect is similar to the equalizing effect of polishing

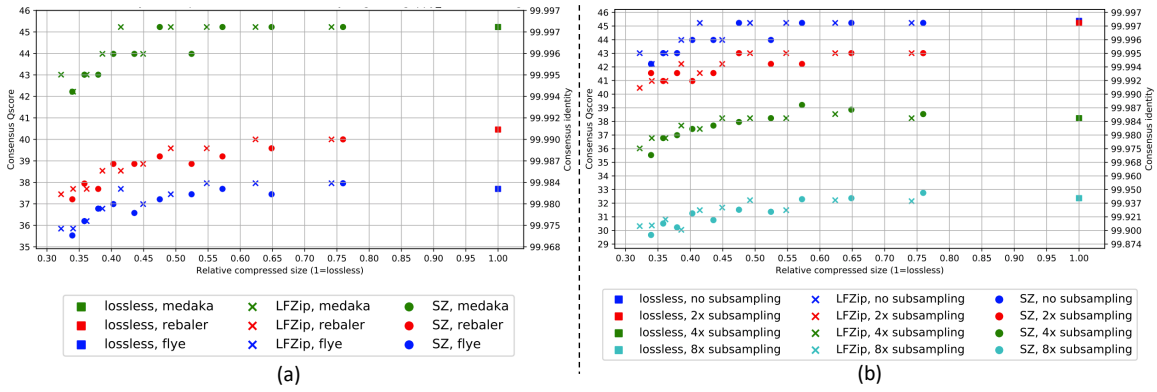


Figure 4.6: (a) Consensus accuracy vs. compressed size after each assembly step (Flye, Rebaler, Medaka) for the *S. aureus* dataset basecalled with guppy_hac. (b) Consensus accuracy (Medaka polished) vs. compressed size for subsampled versions (original, 2X subsampled, 4X subsampled, 8X subsampled) of the *S. aureus* dataset basecalled with guppy_hac. The results are displayed for the losslessly compressed data and the lossily compressed versions with LFZip and SZ (with maxerror 1 to 10). The compressed sizes are shown relative to the VBZ lossless compression size.

applied to different basecallers observed in [54]. We see a similar trend for the other dataset and analysis tools (available on GitHub).

Figure 4.6(b) studies the impact of subsampling to lower read depths on the consensus accuracy (after Medaka polishing) for the *S. aureus* dataset basecalled with guppy_hac. Note that the original dataset has around 80x depth of coverage, so 8x subsampling produces a depth of 10x which is generally considered quite low. We observe that lossy compression has more severe impact on consensus accuracy for lower depths, but 40-50% of size reduction can still be achieved without sacrificing the accuracy. This is again expected because consensus works better with higher depth datasets and is able to correct a greater fraction of the basecalling errors. We see a similar trend for the other dataset and analysis tools (available on GitHub).

Figure 4.7 considers the accuracy of homopolymers (of length 5 to 8) for the Medaka polished assembly of the *S. aureus* dataset basecalled with guppy_hac. We see that the impact of lossy compression is more pronounced for longer homopolymer sequences which are harder for the basecaller and assembly tools to handle, with around 30% size reduction over lossless compression possible with negligible impact

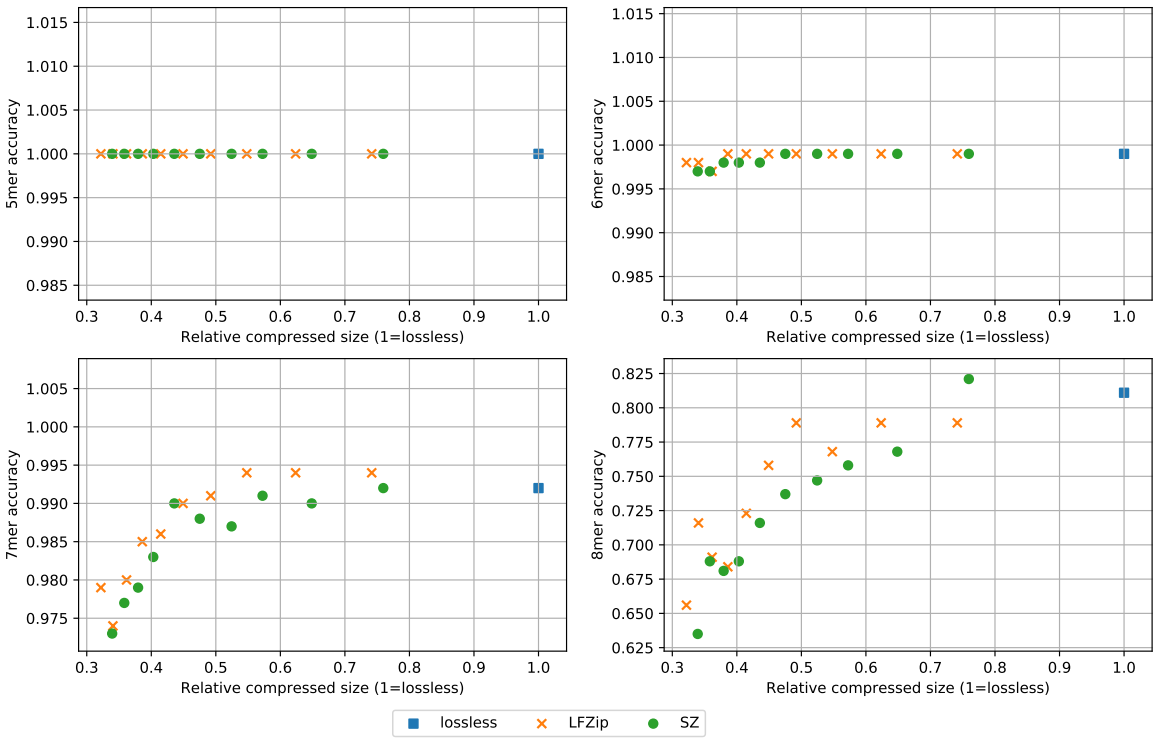


Figure 4.7: Consensus accuracy (Medaka polished) for homopolymer sequences of length 5 to 8 for the *S. aureus* dataset basecalled with guppy_hac. The results are displayed for the losslessly compressed data and the lossily compressed versions with LFZip and SZ (with maxerror 1 to 10). The compressed sizes are shown relative to the VBZ lossless compression size.

on the accuracy. Thus, depending on the requirements, a lower maxerror parameter should be chosen to achieve higher accuracy for the longer homopolymer sequences. We believe it should be possible to overcome this challenge by designing specialized lossy compressors and by training the models in the basecalling and consensus pipeline on the lossily compressed data. We see a similar trend for other subsampling levels and other datasets (available on GitHub).

Overall, we observe that both LFZip and SZ can be used as tools to significantly save on space without sacrificing basecalling and consensus accuracy. The savings in space are close to 50% over lossless compression and 70% over the uncompressed representation. While it is not possible to say with certainty that we don't lose any information in the raw signal that might be utilized by future basecallers, the results for the different basecallers and consensus stages suggest that applying lossy compression (for a certain range of parameters) only affects the noise in the raw signal without affecting the useful components. Finally, the decision to apply lossy compression and the extent of lossy compression should be based on the read depth (coverage), with more savings possible at higher depths where consensus accuracy is the metric of interest.

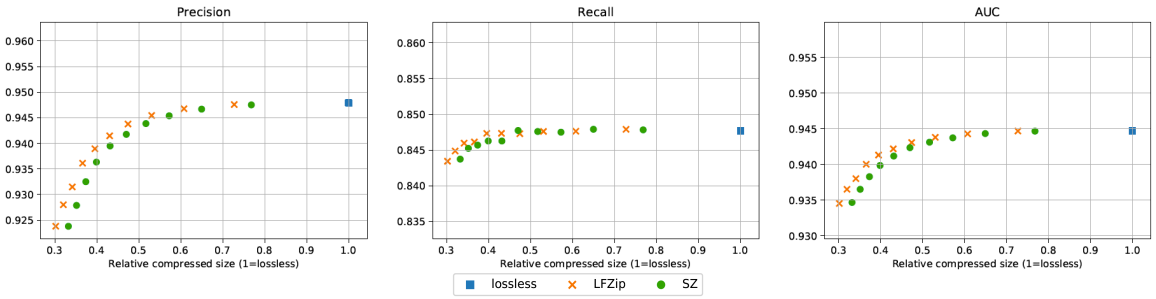


Figure 4.8: Precision, recall and AUC (area under ROC curve) for NA12878 CpG methylation calling using Megalodon. The metrics are computed for per-read methylation calls. For the precision and recall, a probability threshold of 0.5 was used for the predicted methylation probabilities. The results are displayed for the losslessly compressed data and the lossily compressed versions with LFZip and SZ (with maxerror 1 to 10). The compressed sizes are shown relative to the VBZ lossless compression size.

4.4.3 Methylation calling accuracy

Figure 4.8 shows the precision, recall and AUC for CpG methylation calling on the NA12878 *H. sapiens* dataset. Across the >128K reads, there were roughly 670K positive ground truth positions and 560K negative ground truth positions after alignment. Recall that only genomic positions with sufficiently high confidence regarding the methylation status from bisulfite data were considered for the evaluation (∼5.4M positive and ∼4.7M negative sites on genome, counting both strands). The achieved precision, recall and AUC were roughly 0.945, 0.845 and 0.945 respectively. As seen in the figure, the impact of lossy compression is similar to that on basecalling accuracy, with negligible impact for compression gains around 35-40% over lossless compression. This is despite the fact that methylation calling is generally considered a harder problem than basecalling due to the increased resolution needed for it. Further benchmarking using improved benchmark datasets in the future can be performed to strengthen these conclusions.

4.4.4 Time and memory usage

In this section, we provide the time and peak memory usage for lossy/lossless compression, as well as for the steps in the downstream analysis. We focus on the *S. aureus* dataset, with three settings of maxerror (1, 5, 10) for the lossy compressors, and basecalling done with the default basecaller Guppy (high accuracy mode). The three maxerror settings allow comparison across different levels of lossy compression. All experiments were performed on an Ubuntu 18.04.4 server with 40 Intel Xeon processors (2.2 GHz), 260 GB RAM and 8 Nvidia TITAN X (Pascal) GPUs. The default Python version was 3.7.6 (Anaconda).

Table 4.2 shows the time and memory usage for the compression stage. Due to the way the time was measured, for lossy compressors (LFZip, SZ) it includes the time for loading the reads from the original fast5, compressing the raw signal, decompressing, and writing the reconstructed signal to a new fast5. For lossless compressor VBZ, the time includes the time for loading the reads from the original fast5 and compressing the raw signal. Also note that we ran VBZ in the highest compression

Mode	Time (s)	Peak memory (MB)	Average #samples/s
VBZ (lossless)	2135	91.5	1.14M
LFZip (maxerror 1)	4117	64.6	0.59M
SZ (maxerror 1)	814	66.9	2.99M
LFZip (maxerror 5)	3998	67.9	0.61M
SZ (maxerror 5)	773	68.2	3.14M
LFZip (maxerror 10)	3879	67.9	0.63M
SZ (maxerror 10)	697	68.1	3.49M

Table 4.2: Time and peak memory usage for compression+decompression of the *S. aureus* dataset for different compressors and maxerror parameters. The lossless compression time only includes the compression time. The last column shows the average number of raw signal samples handled per second, where the total number of raw signal samples for this dataset is roughly 2.43 billion.

mode (22) since we aimed to compare with the state-of-the-art lossless compressor. The default application with lower compression mode (1) can be significantly faster. All compressors were run on the CPU with a single thread.

From Table 4.2, we see that generally SZ is 5-6 times faster than LFZip, and the compression speed slightly increases with higher maxerror settings. Note that the lossy compression speed is in the order of millions of raw signal samples per second, while the nanopore sampling frequency is 4000 samples/s for DNA pores. Thus, it should be possible to integrate lossy compression in a real-time manner with the sequencing process, especially upon improved integration with the pipeline and further optimization. We note that the peak memory usage is independent of the number of reads, and hence shouldn't be a factor in the scalability of these algorithms.

Table 4.3 shows the time and memory usage for Guppy (high accuracy), which was run using a GPU. We see a very small increase in the time and memory usage for lossy compression as compared to lossless compression, but it is hard to distinguish from experimental noise and we can conclude that the impact of lossy compression the computational requirements for basecalling is relatively minor for this dataset.

Tables 4.4, 4.5 and 4.6 show the time and memory usage for the three assembly/consensus stages: Flye, Rebaler and Medaka, respectively. All three tools were run using 8 CPU threads. We see a small variation in the time and memory usage for

Mode	Time (s)	Peak memory (GB)
VBZ (lossless)	927	6.20
LFZip (maxerror 1)	969	7.42
SZ (maxerror 1)	952	6.67
LFZip (maxerror 5)	1005	6.68
SZ (maxerror 5)	997	6.78
LFZip (maxerror 10)	989	7.14
SZ (maxerror 10)	979	6.97

Table 4.3: Time and peak memory usage for Guppy (high accuracy) basecalling of the *S. aureus* dataset for different compressors and maxerror parameters.

Mode	Time (s)	Peak memory (GB)
VBZ (lossless)	1020	4.58
LFZip (maxerror 1)	1003	4.31
SZ (maxerror 1)	1054	5.12
LFZip (maxerror 5)	1007	4.49
SZ (maxerror 5)	993	4.53
LFZip (maxerror 10)	885	4.47
SZ (maxerror 10)	936	4.05

Table 4.4: Time and peak memory usage for Flye assembly of the *S. aureus* dataset for different compressors and maxerror parameters.

Mode	Time (s)	Peak memory (GB)
VBZ (lossless)	1088	1.04
LFZip (maxerror 1)	1100	1.01
SZ (maxerror 1)	1102	0.97
LFZip (maxerror 5)	1122	1.03
SZ (maxerror 5)	1111	1.04
LFZip (maxerror 10)	1178	1.23
SZ (maxerror 10)	1189	1.19

Table 4.5: Time and peak memory usage for Rebaler consensus of the *S. aureus* dataset for different compressors and maxerror parameters.

Mode	Time (s)	Peak memory (GB)
VBZ (lossless)	109	8.07
LFZip (maxerror 1)	93	8.13
SZ (maxerror 1)	93	8.82
LFZip (maxerror 5)	95	7.66
SZ (maxerror 5)	95	7.34
LFZip (maxerror 10)	98	9.95
SZ (maxerror 10)	98	9.11

Table 4.6: Time and peak memory usage for Medaka polishing of the *S. aureus* dataset for different compressors and maxerror parameters.

lossy compression as compared to lossless compression (particularly at high maxerror), but generally the variation is hard to distinguish from experimental noise and we can conclude that the impact of lossy compression the computational requirements for assembly/consensus is relatively minor.

4.5 Conclusions and future work

In this chapter, we explored the use of lossy compression for nanopore raw data and its impact on the basecalling and consensus accuracy. We found that lossy compression with existing tools can reduce the compressed size by 35-50% over lossless compression with less than 0.2% percent reduction in basecalling accuracy. The impact on consensus accuracy is even lower with less than 0.002% reduction at similar compression levels. Similar conclusions hold across datasets at different depths of coverage as well as several basecalling and assembly stages (with slight variation in the impact due to baseline lossless levels), suggesting that lossy compression with appropriate parameters does not lead to loss of useful information in the raw signal. For datasets with high depth of coverage, even further reduction is possible without sacrificing consensus accuracy. The analysis pipeline and data, partly based on [54] and its addendum, are available online on GitHub along with documentation and can be useful for further experimentation and development of specialized lossy compressors for nanopore raw signal data, which is part of future work. Further experiments on methylation accuracy evaluation and assembly for larger eukaryotic genomes are

also part of future work, contingent upon the availability of improved benchmark datasets.

We believe that further research in this direction can lead to lossy compression algorithms tuned to the specific structure of the nanopore data and the evaluation metrics of interest, leading to further reduction in the compressed size. Further research into modeling the raw signal and the noise characteristics can help in this front. Another interesting direction could be the possibility of jointly designing the lossy compression with the modification of the algorithms in the downstream applications to match this compression. In particular, the current neural network models used in the basecallers can be retrained on the lossily compressed data to further understand the loss in information due to lossy compression. Finally, just as research on impact of lossy compression of Illumina quality scores on variant calling [74, 75] led to Illumina reducing the default resolution of quality scores, it might be interesting to explore a similar possibility for nanopore data by performing the lossy compression or data binning on the nanopore sequencing device itself.

Chapter 5

Concluding Remarks

This thesis considered the problem of compression of raw genomic data. Given the breathtaking growth in the data volume, we saw that general-purpose compressors are no longer sufficient for efficient storage, transfer, and analysis of this data. Thus, we proposed specialized compressors that exploit the structure in the data using an underlying information-theoretic framework. We also explored the use of lossy compression of genomic data to achieve further size reduction by removing irrelevant details and noise while preserving the information we care about. The proposed tools perform well on real-world genomic data, achieving significant reduction in the size while being computationally practical.

Bibliography

[1] Shubham Chandak et al. SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*, 35(15):2674–2676, 12 2018.

[2] Shubham Chandak et al. LFZip: Lossy compression of multivariate floating-point time series data via improved prediction. In *2020 Data Compression Conference (DCC)*, pages 342–351. IEEE, 2020.

[3] Shubham Chandak et al. Impact of lossy compression of nanopore raw signal data on basecalling and consensus accuracy. *Bioinformatics*, 36(22-23):5313–5321, 12 2020.

[4] Ibrahim Numanagić et al. Comparison of high-throughput sequencing data compression tools. *Nature Methods*, 13(12):1005, 2016.

[5] Faraz Hach et al. SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057, 2012.

[6] James K Bonfield and Matthew V Mahoney. Compression of FASTQ and SAM format sequencing data. *PloS one*, 8(3):e59190, 2013.

[7] Łukasz Roguski and Sebastian Deorowicz. DSRC 2-Industry-oriented compression of FASTQ files. *Bioinformatics*, 30(15):2213–2215, 2014.

[8] Łukasz Roguski et al. Fastore: a space-saving solution for raw sequencing data. *Bioinformatics*, 34(16):2748–2756, 2018.

[9] Idoia Ochoa et al. Effect of lossy compression of quality scores on variant calling. *Briefings in Bioinformatics*, 18(2):183–194, 2017.

[10] Greg Malysa et al. QVZ: lossy compression of quality values. *Bioinformatics*, 31(19):3122–3129, 2015.

[11] Shubham Chandak et al. Compression of genomic sequencing reads via hash-based reordering: algorithm and analysis. *Bioinformatics*, 34(4):558–567, 2018.

[12] Peter J. A. Cock et al. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, 2010.

[13] Illumina. Illumina technical note, "Quality Scores for Next-Generation Sequencing", `https://www.illumina.com/documents/products/technotes/technote_Q-Scores.pdf`, .

[14] Illumina. Illumina technical note, "Reducing Whole-Genome Data Storage Footprint", `https://www.illumina.com/documents/products/whitepapers/whitepaper_datacompression.pdf`, .

[15] R. Long et al. Genecomp, a new reference-based compressor for sam files. In *2017 Data Compression Conference (DCC)*, pages 330–339, April 2017.

[16] Michael A. Eberle et al. A reference data set of 5.4 million phased human variants validated by genetic inheritance from sequencing a three-generation 17-member pedigree. *Genome Research*, 27(1):157–164, 2017.

[17] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

[18] Tomasz M Kowalski and Szymon Grabowski. PgRC: pseudogenome-based read compressor. *Bioinformatics*, 36(7):2082–2089, 12 2019.

[19] Sebastian Deorowicz. Fqsqueezer: k-mer-based compression of sequencing data. *Scientific reports*, 10(1):1–9, 2020.

[20] Claudio Alberti et al. An introduction to MPEG-G, the new ISO standard for genomic information representation. *bioRxiv*, 2018.

[21] Idoia Ochoa et al. Effect of lossy compression of quality scores on variant calling. *Briefings in bioinformatics*, 18(2):183–194, 2016.

[22] Allen Gersho and Robert M Gray. *Vector quantization and signal compression*, volume 159. Springer Science & Business Media, 2012.

[23] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[24] EH Bristol. Swinging door trending: Adaptive trend recording? In *ISA National Conf. Proc., 1990*, pages 749–754, 1990.

[25] George Edward Williams. Critical aperture convergence filtering and systems and methods thereof, July 11 2006. US Patent 7,076,402.

[26] EVSystems Data Solutions. `https://www.evsystems.net/knowledge_base`.

[27] OSIsoft. `https://livelibrary.osisoft.com/LiveLibrary`.

[28] Sheng Di and Franck Cappello. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 730–739. IEEE, 2016.

[29] Dingwen Tao et al. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1129–1139. IEEE, 2017.

[30] Xin Liang et al. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447. IEEE, 2018.

[31] Xin Liang et al. An efficient transformation scheme for lossy data compression with point-wise relative error bound. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 179–189. IEEE, 2018.

[32] Sriram Lakshminarasimhan et al. ISABELA for effective in situ compression of scientific data. *Concurrency and Computation: Practice and Experience*, 25(4):524–540, 2013.

[33] Zhengzhang Chen et al. NUMARCK: machine learning algorithm for resiliency and checkpointing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 733–744. IEEE Press, 2014.

[34] S Edward Hawkins III and Edward Hugo Darlington. Algorithm for compressing time-series data. *NASA Tech Briefs*, 2012.

[35] Naoto Sasaki et al. Exploration of lossy compression for application-level checkpoint/restart. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 914–922. IEEE, 2015.

[36] Douglas O'Shaughnessy. Linear predictive coding. *IEEE potentials*, 7(1):29–32, 1988.

[37] Webp. `https://developers.google.com/speed/webp/`. Accessed: 2019-10-22.

[38] Christian Feller et al. The VP8 video codec-overview and comparison to H. 264/AVC. In *2011 IEEE International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*, pages 57–61. IEEE, 2011.

[39] Mohit Goyal et al. DeepZip: Lossless Data Compression Using Recurrent Neural Networks. In *2019 Data Compression Conference (DCC)*, pages 575–575. IEEE, 2019.

[40] Qian Liu et al. DecMac: A Deep Context Model for High Efficiency Arithmetic Coding. In *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 438–443. IEEE, 2019.

[41] Matteo Pagin and Maurits Ortmanns. A neural data lossless compression scheme based on spatial and temporal prediction. In *2017 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pages 1–4. IEEE, 2017.

[42] Ilya Grebnov. BSC. `http://libbsc.com/`, 2015.

[43] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[44] Allan Stisen et al. Smart devices are different: Assessing and mitigating mobile sensing heterogeneities for activity recognition. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140. ACM, 2015.

[45] Individual household electric power consumption Data Set. `https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption`. Accessed: 2019-10-22.

[46] Philip Schmidt et al. Introducing WESAD, a Multimodal Dataset for Wearable Stress and Affect Detection. In *Proceedings of the 2018 on International Conference on Multimodal Interaction*, pages 400–408. ACM, 2018.

[47] Ramon Huerta et al. Online decorrelation of humidity and temperature in chemical sensors for continuous monitoring. *Chemometrics and Intelligent Laboratory Systems*, 157:169–176, 2016.

[48] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[49] Günther Foidl. Swinging Door. `https://github.com/gfoidl/DataCompression`.

[50] Miten Jain et al. The Oxford Nanopore MinION: delivery of nanopore sequencing to the genomics community. *Genome biology*, 17(1):239, 2016.

[51] Miten Jain et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature biotechnology*, 36(4):338, 2018.

[52] Scott Gigante. Picopore: a tool for reducing the storage size of oxford nanopore technologies datasets without loss of functionality. *F1000Research*, 6, 2017.

[53] Xin Liang et al. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447. IEEE, 2018.

[54] Ryan R Wick et al. Performance of neural network basecalling tools for Oxford Nanopore sequencing. *Genome biology*, 20(1):129, 2019.

[55] Guillermo Dufort y Álvarez et al. ENANO: Encoder for NANOpore FASTQ files. *Bioinformatics*, 05 2020. btaa551.

[56] Franka J Rang et al. From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy. *Genome biology*, 19(1):90, 2018.

[57] Alex Graves et al. Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on Machine learning*, pages 369–376, 2006.

[58] Mikhail Kolmogorov et al. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology*, 37(5):540–546, 2019.

[59] Yu Lin et al. Assembly of long error-prone reads using de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016.

[60] Sergey Koren et al. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome research*, 27(5):722–736, 2017.

[61] Robert Vaser et al. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome research*, 27(5):737–746, 2017.

[62] Nicholas J Loman et al. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature methods*, 12(8):733–735, 2015.

[63] Jared T Simpson et al. Detecting DNA cytosine methylation using nanopore sequencing. *Nature methods*, 14(4):407–410, 2017.

[64] Qian Liu et al. Detection of DNA base modifications by deep recurrent neural network on Oxford Nanopore sequencing data. *Nature communications*, 10(1):1–11, 2019.

[65] Peng Ni et al. DeepSignal: detecting DNA methylation state from Nanopore sequencing reads using deep-learning. *Bioinformatics*, 35(22):4586–4595, 2019.

[66] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[67] Allen Gersho and Robert M Gray. *Vector quantization and signal compression*, volume 159. Springer Science & Business Media, 2012.

[68] Justin M Zook et al. Extensive sequencing of seven human genomes to characterize benchmark reference materials. *Scientific data*, 3(1):1–26, 2016.

[69] Haotian Teng et al. Chiron: translating nanopore raw signal directly into nucleotide sequence using deep learning. *GigaScience*, 7(5), 04 2018. giy037.

[70] Jingwen Zeng et al. Causalcall: Nanopore basecalling using a temporal convolutional network. *Frontiers in Genetics*, 10:1332, 2020.

[71] Samuel M Nicholls et al. Ultra-deep, long-read nanopore sequencing of mock microbial community standards. *Gigascience*, 8(5):giz043, 2019.

[72] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[73] ENCODE-Project-Consortium et al. An integrated encyclopedia of dna elements in the human genome. *Nature*, 489(7414):57–74, 2012.

[74] Y William Yu et al. Quality score compression improves genotyping accuracy. *Nature biotechnology*, 33(3):240–243, 2015.

[75] Idoia Ochoa et al. Effect of lossy compression of quality scores on variant calling. *Briefings in bioinformatics*, 18(2):183–194, 2017.