Imperial College London

Final year project

Novel Parallelized Genomics Compression

Author

Samuel Wong

Supervisor

Wayne Luk

James Arram

Abstract

This report presents a new approach for highly parallel compression and decompression of genomics data. There are four main contributions. First, an effective lossless compression algorithm for genomic data in the FASTA and FASTQ formats is developed. Secondly, the compression algorithm is dissected and parallelized on both the software and hardware fronts, resulting in a speedup of 20-fold compared to other compression tools with similar compression ratios. Thirdly, modifications are made and proposed to reduce the memory usage of the compression tool. Finally, evaluation of the proposed approach by benchmarking it against other existing compression tools currently used in genomics research.

Acknowledgements

I would like to thank my supervisor Professor Wayne Luk for first of all allowing me the opportunity to pursue a topic that I find both interesting and challenging, and also for his guidance and support throughout the project.
Secondly, I am immensely grateful to James Arram who has provided invaluable feedback and suggestions throughout the development of the compression tool. The compression tool would not have obtained the promising results it would have without his existing work on the compression of sequencing data and its corresponding acceleration.

Contents

1 Introduction

1.1 Motivation

The human pursuit of obtaining knowledge consistently revolves around the understanding of our surrounding environments, ranging from the scales of the cosmos to the study of minuscule elemental particles. However, rather than looking outwards, parts of the scientific community looks inwards and is devoted to studying ourselves - the human body and human mind. One of the keys to understanding how the human body works is Deoxyribonucleic Acid, or *DNA*, which carries the genetic instructions that dictate the function and development of all known forms of life.

DNA sequencing is the process of identifying the order of nucleotide bases which are the building blocks in a DNA molecule. Knowledge of such sequences has been critical to advances in research from a multitude of scientific disciplines including but not limited to medicine, molecular biology, and forensics. For example, medical researchers have gleaned a great deal of insight into genetic diseases and can determine risks of genetics diseases in patients with a high degree of certainty. Thus the impact of studies on genomic sequences cannot be understated which is demonstrated by steadily increasing funding into genomic research[12].

Throughout decades, the methods for DNA sequencing have been improved by multiple orders of magnitude in terms of both time and cost. Modern sequencing technology, commonly known as *Next-generation Sequencing*, can analyze DNA samples and detect the constituent nucleobases at an incredible speed. The rapid increase in computational power and speed in sequencing technology has led to the analysis of DNA sequences and entire genomes of a wide variety of different species including *Homo sapiens* ourselves, potentially even unraveling centuries old mysteries such as how human beings came to be. Nonetheless, the evolution of sequencing techniques has posed a different problem entirely - that the tremendous throughput of sequencing machines are improving at a rate greater than Moore's law, thus leading to the problem of running out of storage for the output sequencing data.

Currently, some bioinformatics researchers have adopted the utilization of general purpose compression utilities to perform compression on the genomics data due to its accessibility and ease of use. General purpose compression algorithms such as *bzip2* could reach a compression rate of approximately 2- to 4-fold depending on the characteristics of the dataset. However, this is still an unsatisfactory solution, and thus, the need for the development of a domain-specific compression algorithm which produces better results when applied specifically to genetic data, has arisen.

In addition to compression rate, the compression and decompression speed is crucial to the viability of a compression tool as it is meaningless to develop a compression utility which would take longer to compress the output data than the time it takes the sequencing machines to output data. Even with computers packed with computational power, compression of a single genomics data file may take up to several hours depending on the size of the data. This has sparked the need for hardware acceleration of compression tools to combat the problem.

To alleviate or solve the aforementioned problem of the storage of the torrent of new sequencing data, a well-designed compression algorithm is required, which should be combined with proper optimization and acceleration on both software and hardware fronts to ensure it does not create a problem in itself.

1.1.1 Aims for the compression utility

To begin developing an algorithm for the compression of genomics data, the requirements and objectives should first be discussed.

Firstly, the compression tool has to be *lossless*, which requires that the compression to be completely reversible without any loss of information. In mathematical terms, the compression should be bijective, which implies that every possible input has a unique corresponding output and every compressed file can be uniquely decodable.

Secondly, the targeted domain of the compression tool will be the formats **FASTQ**, which are the most pervasive genomics data formats currently in circulation. Moreover, as a consequence of the relationship between **FASTQ** and **FASTA**, the compression tool would easily be adapted to support the **FASTA** format as well.

Thirdly, the main objective of a domain specific compression tool is to achieve better compression results than current compression tools used by genetic researchers. The results will be evaluated using four components:

- **A1** - the *compression ratio*, which is the ratio between the size of the compressed file and the original file, hereafter referred to as the compression ratio;
- **A2** - the *compression speed* and *decompression speed*, which is measured by the time for **compress** to compress and decompress the given file respectively; and
- **A3** - the *maximum memory usage*, which is the peak virtual Random Access Memory used.

The compression ratio is critical due to the advances in next generation sequencing technology in the past decade that has surpassed the evolution of computer storage in the same timeframe.

The compression speed should be substantially quicker than the sequencing technology for the same amount of data so that the usage of the tool does not create a bottleneck in the storage and transfer of genomics data. Similarly, the decompression speed should be within acceptable ranges in order to not impede the access of compressed genomics data by geneticists. A more quantitative goal for the compression and decompression speeds would be to able to compress and decompress the complete human genome within 20 minutes.

Moreover, the compression tool should not be utilizing huge amounts of memory as it should be able to run on computers used in genomics research labs. A more challenging goal would be to allow the average computer with about 16 gigabytes of memory to run the compression tool without huge time penalties from page swaps due to insufficient memory.

Selective Decompression

In many cases, it is immensely helpful if the bioinformaticians studying the compressed data are able to selectively decompress sections of the sequence without completely decompressing the entire file. Of course, enabling this will end up limiting our choices of algorithms. If we limit ourselves to fixed-length code, where each FASTQ entry is encoded into the same number of bits, it is possible for the program to calculate the required offset and decompress particular sections of interest. On the other hand, for example dictionary codes (such as Lempel-Ziv's algorithms and their variations) which decoding depends on the information previously decoded and stored in the dictionary cannot be utilized without some sort of modification.

Thus we will discuss both codes that allow for selective decompression and those which do not when developing the compression algorithm in the following sections. Both of these choices will be implemented and we will allow the users to decide on the tradeoff between the compression ratio and the ability for random access through selective decompression.

Human genome sequences

Although the compression algorithm should be able to compress any sequence in the correct format, particular emphasis is put on the compression of human genome sequences, where the length and variety of human sequences present a challenge to the scientists trying to study and understand the purpose of various sections of the human genome. While the compression tool will not be made specifically for human sequences, we will attempt to tailor fit the compression algorithm by using concepts and algorithms that will be more effective when compressing high quality, high coverage and long sequences. These high quality sequences typical of the output of newer versions of next generation sequencing machines are the most sought after genomics data used in clinical and biomedical research settings.

1.2 Contributions

In this work we present the following contributions:

- **C1** - We develop a novel lossless compression tool for genomic data in the FASTA and FASTQ formats. The compression tool yields up to a 6-fold decrease in file size, and is especially effective given high quality, high coverage and long sequences such as the human genome. We develop novel methods to compress the metadata (also known as the sequence identifiers) and the quality scores:
  - **C1.1** Firstly, the metadata is tokenized. Then, these tokens are analyzed and classified into archetypes with common behaviour. Depending on the type, the token is then encoded into minimal amounts of memory using a highly-adaptive novel algorithm to find the optimal encoding for each field based on the characteristics of the archetype and the values of the input metadata. The compression of the metadata achieves greater than a 10-fold decrease in file size.
  - **C1.2** The method to compress quality scores involves initially the quality scores are translated into run-length encoding symbols. The quality scores are then grouped in order to reduce the entropy, and the finally compressed via arithmetic encoding;
- **C2** - We optimize the compression tool by acceleration using novel greedy algorithms in determining the encoding of the quality scores and metadata, multithreading techniques and OpenMP to improve the performance in terms of compression and decompression speeds. The sequence compression is further accelerated using Maxeler's dataflow computing engine. The acceleration reduces the overall compression time of a complete human genome (approx. 19 gigabytes) to 270 seconds, reduced from over 7000 seconds which is required by compression tools that yield the same compression ratio;
- **C3** - We employ various techniques to reduce the maximum memory usage, such as subdividing the input file to be compressed and reducing the size of the reference index and suffix array in the reference-based sequence compression stage by sampling during precomputation of the references; and
- **C4** - The compression tool presented is evaluated by benchmarking against other existing compression tools. Six FASTQ datasets are chosen, containing various species, read lengths and read counts. The six compression utilities, of which two are general purpose compression utilities and the other four domain-specific compression tools designed for FASTQ data, is then benchmarked using the following criterion: (1) compression rate, (2) compression speed, and (3) decompression speed.

1.3 Report structure

The compression algorithm can be visualized using figure 1 on the next page. In the next chapter, we present the user with a multi-disciplinary overview of the background information, and presents the sequence encoder based on the work done on the sequence compression by James Arram et al. In chapter 3, the metadata compression encoder is presented in its entirety. In chapter 4, the quality scores are analyzed and then the quality score compression encoder, which consists of the modified run-length encoding, ternary grouping and arithmetic encoding stages, is developed. In chapter 5, we complete the main compression algorithm by describing the strand encoder followed by the *bzip2* compression of the compressed sequences and quality scores and then finally combining the compressed components into one archive. Chapter 6 presents novel modifications to the algorithm to enable and exploit the paralellism at different stages in the compression pipeline. Finally, chapter 7 evaluates our compression tool with respect to other compression tools currently used in genetics research.

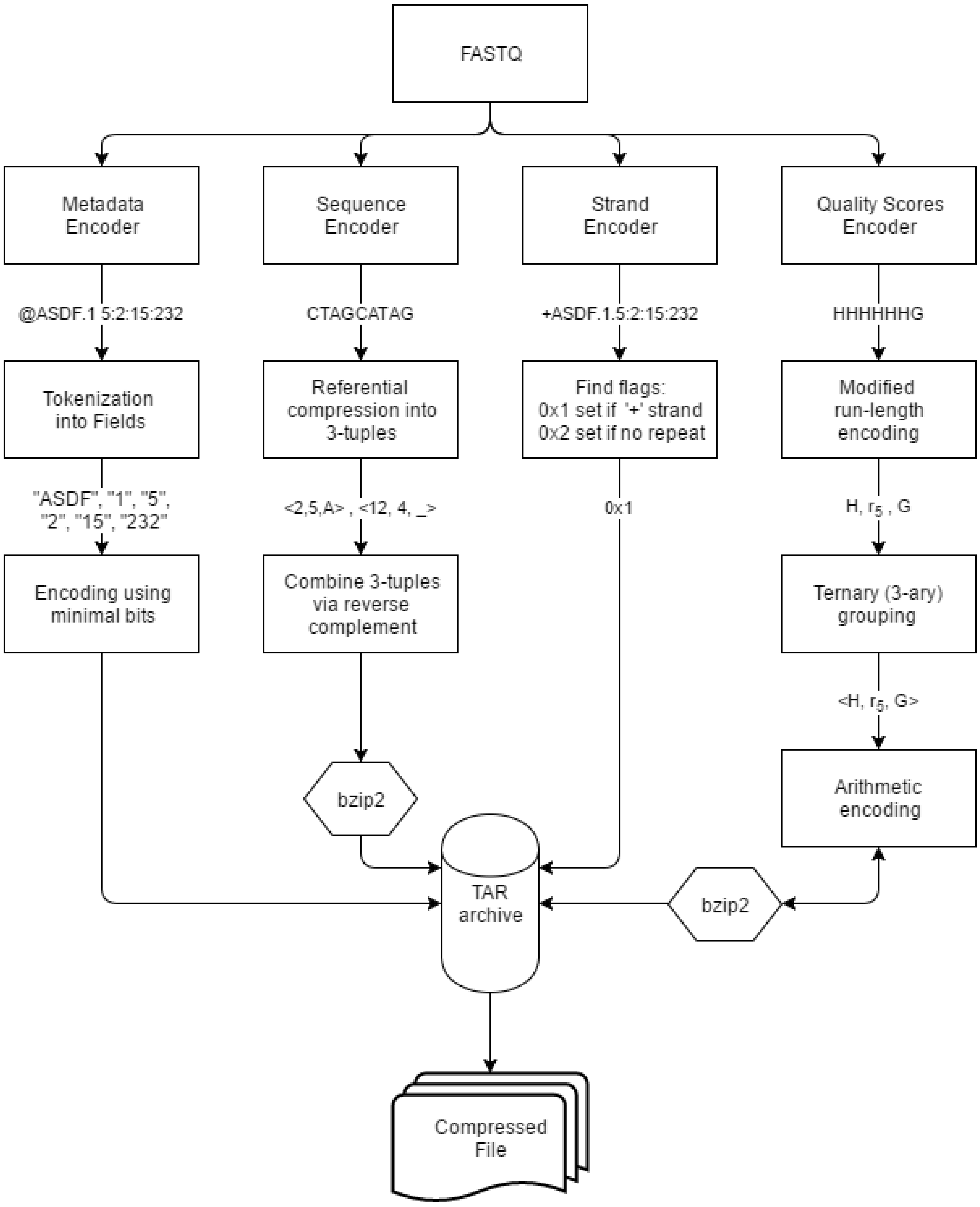


Figure 1: Overall pipeline of compression algorithm

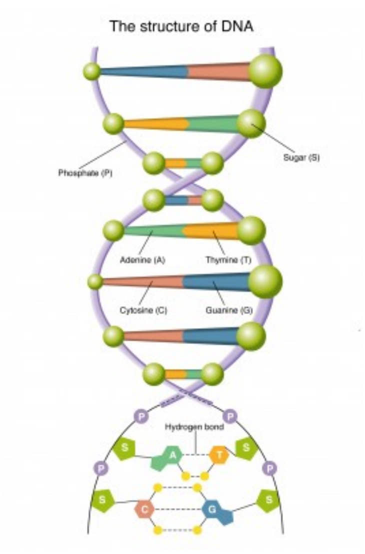
2 Background and Related Work

In this section, the reader is presented with background information that forms the basis of the work on the compression tool performed later on. A brief introduction of genomics and the problem of the storage of the exponentially increasing amounts of genomics data is given. Then, a broad range of compression algorithms are described. Moreover, overviews of the selected compression tools currently used by bioinformatic researchers are presented, which will later be used for the evaluation of the developed compression tool. Finally, the sequence compression algorithm incorporated into the compression tool is detailed.

Chapter 6 proposes novel methods of parallelizing and accelerating the majority of the stages in the compression and decompression pipeline. Finally, chapter 7 presents an evaluation of our compression tool with respect to other compression tools that are currently in use in the field of bioinformatics.

2.1 Genomics

Genomics is a scientific discipline that sequences, assembles, and analyzes the function and structure of the genome - the complete set of Deoxyribonucleic Acid or DNA in an organism. DNA are molecules that carry instructions that exists in all known living organisms and is the basis which allows the organisms to develop and function, and is thus essential to all forms of life. The human genome contains 23 chromosomes, where each chromosome contains 48 to 250 million base pairs, which gives a total of over 3.2 billion nucleobases.[5]



There are four nucleobases in DNA, which are Adenine, Cytosine, Guanine and Thymine. In Ribonucleic Acid, hereafter referred to as *RNA*, the nucleobase Thymine is replaced with Uracil. The double helix structure of DNA is formed by the two pairs of hydrogen bonding which are Adenine-Thymine and Cytosine-Guanine. These nucleobases dictate the instructions of the productions of proteins, which allow the human body to function at a microscopic level. As DNA replication is not a flawless process, mutations sometimes occur, which is indicated by a difference in the sequence of nucleobases. Although most of the times these mutations may be benign, in some cases it may cause disruptions in functionality of the organism which range from relatively manageable conditions such as asthma to incurable (at present time) autoimmune diseases such as multiple sclerosis and neurodegenerative disorders such as Huntington's disease.[17]

Through studying the human genomes, medical researchers have found the specific mutations in particular regions in the DNA that correspond the diseases, which undoubtedly is a major step in finding a cure for these genetic disorders.

2.2 Genomics Sequencing

In 1977, Frederick Sanger et al developed a DNA sequencing method which would become the most pervasive sequencing method for the next 25 years.[23] Nonetheless, the ever increasing demand for sequencing methods have led to the development of "Next-generation sequencing" methods[6] which provide high throughput and low cost technologies which have supplanted Sanger's method as the primary tools for DNA sequencing.

As seen from the table above, next-generation sequencing techniques such as the three listed above are all are multiple orders of magnitudes better than Sanger's method in both time and cost. These next-generation sequencing methods utilize different approaches to genome sequencing; nonetheless, Illumina's machines which deploys the sequencing by synthesis technique account for 70% of the market for

Table 1: Comparison of Sanger's and Next-generation sequencing methods

Method	Time / 1 million bases	Cost / 1 million bases (in US$)
Single-molecule real-time sequencing (Pacific Biosciences)[21]	10 seconds	$0.13 to $0.60
Sequencing by synthesis (Illumina)	1.44 seconds	$0.05 to $0.15
Sequencing by ligation (SOLiD)[26]	5.04 seconds	$0.13
Chain termination (Sanger)	15 days	$2400

genome-sequencing machines and 90% of all DNA data produced, and the listed company has a market capitalization of over USD $20 billion.[15]

Furthermore, this increase in the abilities of the sequencing techniques have surpassed the advances in computer memory predicted by Moore's law. Moore's law predicts that the number of transistors in dense integrated circuits doubles every two years. However, from the graph below it can be seen that the growth in sequencing technology has surpassed the model of Moore's law over the same period of time. This poses a huge problem as within the next decade, the sequenced data may no longer be stored unless previous data is discarded.

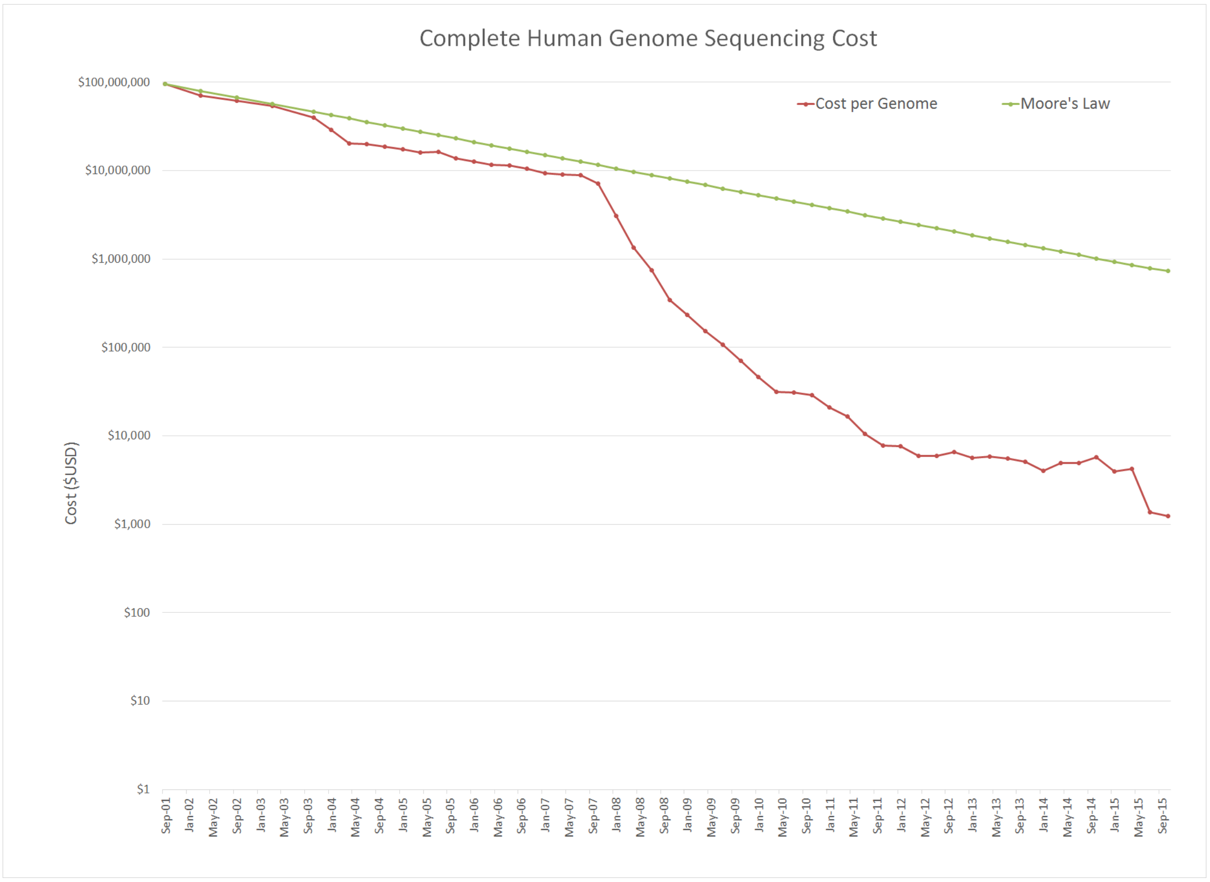


Figure 2: Costs of sequencing the complete human genome over time[30]

Currently, the sequenced nucleotides are output in ASCII characters, and thus each base would use up 1 byte of memory. One single next generation sequencing machine would output up to 1MB of nucleobases per second, which compounded with the wealth of associated data such as the sequence identifiers and the quality scores, fill up huge amounts of memory in a very short span of time.

2.3 Formats of sequencing data

To successfully develop compression algorithms targeted at the genomics data, it is essential to understand the common formats used for storing and distribution the data. The two of the most commonly utilized formats are FASTA and FASTQ which will be described below and are the file formats that the compression algorithm will revolve around.

2.3.1 FASTA format

Originated from the FASTA software package[31], the FASTA format has become a standard in the field of bioinformatics. It is one of the simplest formats and contains a header or description line which is then followed by the protein or DNA sequence. The read sequence is usually the set of nucleotides *A, C, T, G* (and occasionally *U* for RNA sequences). Furthermore, if the sequencing process is unable to discern a certain base call, it will emit a symbol representing the unknown nucleotide which is commonly *N*. Each FASTA sequence is delimited by the metadata corresponding to the later sequence. The metadata contains various information about the read such as but not limited to the sequencing machine's identifier, the position of the read and its length. Note that the FASTA format strongly suggests that each line should be no longer than 120 characters.

```
>SRR327342.2 HWUSI-EAS574_98029123 length=138
NGATGTTTGAATGCACTTTATTTATGGGAAAGAGAAAGCTTTGTCTNACNNNNNNNNNAGAAANTGCGAAAGAGNANAANNNNNNANNGAGNGNG
```

2.3.2 FASTQ format

The FASTQ[11] format is originally developed at the Wellcome Trust Sanger Institute to incorporate additional information such as the quality scores and which strand the DNA is sequenced from to the information found in a FASTA sequence, and since has become the de factor standard output format of next-generation sequencing technology.

```
@ERR161544.18 B81CBVABXX:5:1:1574:2113#TNNNNNNN/1
TCCACCAGTAGGCATCTGTAGCGGACATCTGTTACTTCTGTCTGCCCAGCAACGATTCCCTCTTCTTCC
AGTAATAGCACACAATTTACTTGGTGTAGCT
+ERR161544.18 B81CBVABXX:5:1:1574:2113#TNNNNNNN/1
HHHHHHGHHDHHEHHHHFFHHHHHHHHHHHHHHHHHHHHHHHHHHHHHBHHHEHHHHCHEEDF@HFAHHEE?G
DA=BC@EBFEFHFEHEHHHEEHEHFHDB@=C
```

Each entry in the FASTA format contains four lines:

1. The header containing *metadata* including the sequence identifier and optionally the description of length, genome and et cetera similar to FASTA's header;
2. The DNA *sequence*, once again containing the nucleotides and occasionally the symbols corresponding to unknown bases;
3. The *strand*, which is '+' or '-' symbol denoting the strand of DNA the sequence is retrieved from, *optionally* followed by the exact replica of the metadata on the first line; and lastly
4. *Quality scores* corresponding to the each nucleotide in the read, and thus the quality scores would have an identical length to the sequence in the second line.

Note that FASTQ does not impose FASTA's length limit on each line - and thus there will be exactly four lines per read.

The quality scores are measures of the sequencing machine's evaluation of the probability of the corresponding base pair in the sequencing line being incorrect. Before the inclusion of quality scores, scientists would often find themselves with discrepancies between overlapping DNA sequence reads, which then would require tedious determination of the correct base call. With the inclusion of quality scores, the process of solving conflicts between reads can be automated.

The most commonly utilized method to assess the reliability of each base pair detected is the Phred quality score, which is given by

$$Q_{Phred} = -10 \cdot log_{10}P$$

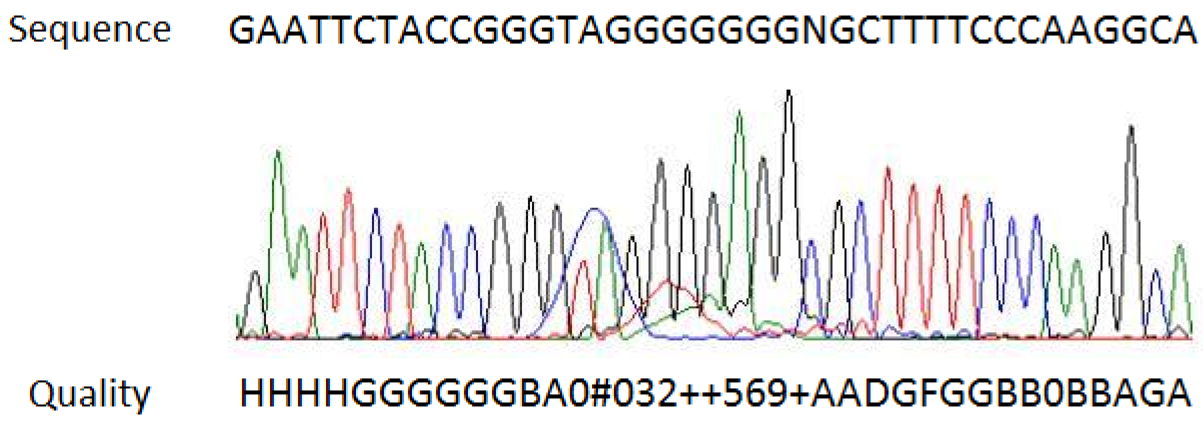


Figure 3: The generation of quality scores during DNA sequence tracing. The different colored peaks correspond to the detected signal of a base pair. The sequence takes the strongest signal, which may be incorrect when the signal strengths of different bases are similar. Thus, the quality scores depict the probability of that interpreted base being correct (for example in Illumina's Casava v1.8 and later, textitGs and Hs which denote high confidence in the correct nucleobase being sequenced, while the hash symbol (#) denotes very low confidence in the resultant base call.

where P is the probability of the base being incorrectly identified. Quality scores from most platforms offset and then encoded using ASCII characters with the range from 33 to 75, i.e.

$$!''\#\$\%\&'()*+,-./0123456789:;<=>?@ABCDEFGHIJK$$

The reason for this offset is due to the fact that ASCII characters with lower values may not be printable.

2.4 Data Compression

Data compression involves encoding information using fewer bits than the original input. There are two major types of compression - lossy and lossless.

Lossy compression reduces the number of bits required by removing similar or redundant information. For example, by recognizing the fact that human eyes can only receive a limited range of colors and cannot differentiate between tiny variations in color, lossy compression algorithms may resort to encoding distinct but similar colors into the same color in order to reduce the comlexity and thus the size of the file.

Lossless compression, on the other hand, exploit statistical redundancies to represent data and is reversible - hence no information is lost. For example a character sequence **AAAABBBBBB** could be encoded via run-length encoding into **4A6B**, from which the original data could easily be constructed.

As the genomics data we aim to compress would undoubtedly cause issues with bioinformaticians if the base pairs decoded are not equal to those encoded, the compression algorithm we propose only utilizes lossless compression techniques.

Obviously, for a lossless algorithm the original plaintext and the compressed files must form a bijective mapping. Thus, for a compression algorithm to decrease the size of some files, the same compression algorithm will increase the file size of others. Therefore, it is necessary to understand the input data and find suitable domain-specific encoding methods that will yield decreased file size for our targeted inputs, which in this case is genomic sequencing data.

(a) Original image

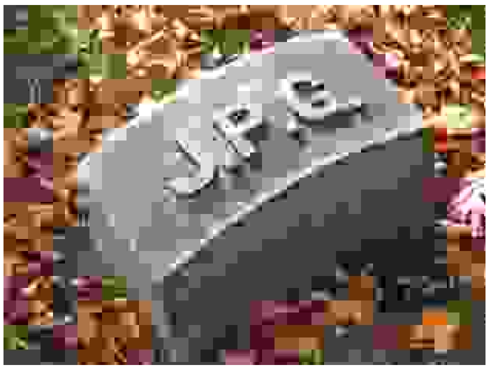

(b) Compressed

Figure 4: Lossy compression of image using JPEG

2.4.1 Run-length encoding

Run-length encoding is a lossless data compression algorithm that encodes consecutive appearances of the same character into tuples <*chr, len*> where *chr* is one single character and *it* is the count of the "run" length. As an example, the string `HHHHFFHHHHHHHHHHHHHHHHHHHHHHHHHBHHH` would be encoded as `<H,4>, <F,2>, <H,25>, <B,1>, <H,3>`. These tuples are then serialized and written to the compressed output file.

2.4.1.1 Representing the characters

Most modern run-length algorithm supports the complete UTF-8 character set, which requires 8-bits to encode. However, the encoding can be modified to only support selected characters if the allowed characters can be predetermined. Then, the number of bits required to losslessly encode n characters is $\lceil log_2 n \rceil$.

2.4.1.2 Representing the run-length

The number of bits to encode the run-length will determine the efficiency of the run-length encoding. If too many bits are allocated for each run-length, then there would be wasted bits for each encoded tuple. However, if there are too few bits allocated, then tuples of run-length l would have to be split into $\lceil \frac{l}{2^b} \rceil$ where b is the number of bits allocated. Thus it is essential to fit the encoding scheme according to the targeted data if possible.

As a more concrete example, we have a quality score string with allowing 45 input characters (6 bits to encode) and setting aside 3 bits for run length:

Run	Character	Character (encoded)	Run-length	Run-length (encoded)
1	H (39)	100111	3	010
2	I (40)	101000	2	001
3	H (39)	100111	3	010
-	STOP (43)	101010		

Table 2: Run-length encoding example

Which finally gives the bit encoding `100111010101000001100111010101010`. The original size using UTF-8 encoding was 64 bits (8 characters * 8 bits per character) while the compressed size is 33 bits, giving a compression ratio of 51.6%.

Note that as there are no run lengths of 0, we will encode $length - 1$ instead to allow high run-lengths to be encoded in less symbols.

Entropy encoding

Entropy encoding is a data compression scheme that utilize the information about the probabilities of the input symbol/characters in order to generate an optimal encoding.

Definition 2.1. *Entropy* is a measure of disorder from an information source, and is the expected value of the information contained in a given message.

The entropy of an input source can be calculated as follows:

$$H = -\sum_{a \in \mathbb{A}} p_a \cdot log_b p_a$$

where H is the entropy, $\mathbb{A}$ is the number of symbols, p_a is the probability of a symbol $a \in \mathbb{A}$ occurring in the input, and b is the base of the output encoding.

According to *Shannon's source coding theorem*[24], the optimal code length for a symbol a and an output base b is $-p_a \cdot log_b p_a$. It can be easily verified that entropy encoding's worst case scenario is when the probability distribution of the symbols is uniform, while the best case scenario is the opposite where the probabilities are highly imbalanced (imagine if the probability of one symbol is 100%). Then the entropy is actually the theoretical expected minimum number of bits per input symbol! The lower the entropy, fewer bits are *expected* to be used to encode the same amount of symbols, and thus the higher the compression rate. The two most commonly used entropy encoding algorithms, Huffman encoding and Arithmetic encoding, will be explored below.

2.4.2 Huffman encoding

Huffman encoding generates a variable-length prefix free code based on the (estimated) probability or frequency of input symbols. Based on a simple algorithm, each symbol can be assigned one node in a binary tree which represents its corresponding encoded bits. The algorithm to construct the tree takes `O(n^2)` time where n is the number of symbols. If the symbols and the frequencies are sorted, this can be reduced to `O(n)`.

The algorithm to construct the tree is as follows. Initialize all the symbols into leaf nodes. Take the two nodes with the least frequency/probability and add it to a parent node which has the frequency of the sum of its child nodes. Repeatedly select the two nodes with no parents that have the least frequency until there is only one such node left - which becomes the root node of the binary tree.

After the tree is constructed, each symbol will have a corresponding encoding that forms a prefix code. As no codeword in a prefix code is the prefix of another, then the bits are uniquely decodable. Starting from the head node and reading one bit at a time, the decoding algorithm will find the corresponding symbol when it reaches the leaf nodes.

Due to the average codeword length of Huffman codes being within an acceptable margin within the entropy, which is the theoretical lower bound

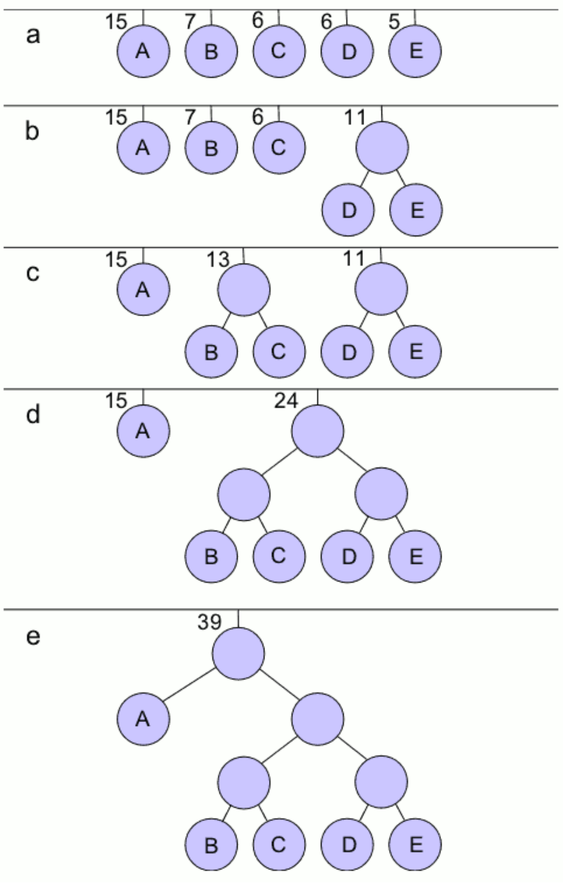


Figure 5: Huffman encoding example[1]

[1]"Huffman Encoding" by Andreas.Roever is licensed under CC BY-SA 3.0

for average codeword length, and coupled with its
ease of implementation and deployment, the Huffman code has been widely utilized. Nonetheless, there are a few cases where the compression is suboptimal and the average codeword length is much greater than the entropy. One of the cases is when the probability of a symbol exceeds 0.5, which makes the upper limit of the inefficiency unbounded as the optimal number of bits for this symbol is less than 1.[25] Furthermore, Huffman codewords must be of integer length as it is impossible to encode a fraction of a bit. Thus, the probabilities of the symbols are not represented optimally as it should be.

2.4.3 Arithmetic encoding

Arithmetic coding[14] is another frequently used form of entropy encoding. Unlike Huffman encoding, rather than treating each symbol separately and individually replacing them with a codeword, arithmetic coding encodes the entire message into a single number.

The naive adaptive form of arithmetic encoding initializes the frequencies of all the symbols to 1. Then, the main loop initializes two variables, *high* and *low* which are set to 1 and 0 respectively. Now, the region between high and low are subdivided into subregions proportional to the symbol frequencies. Then, we set low and high to the boundaries of the subregion corresponding to the symbol currently encoded. This is repeated until the end of the message. For example, we have four symbols 'A', 'B', 'C' and '`<EOF>`' with frequencies 6, 2, 1 and 1 respectively. The `<EOF>` symbol represents the end of the particular message. As an example, the message being encoded is 'AC`<EOF>`'.

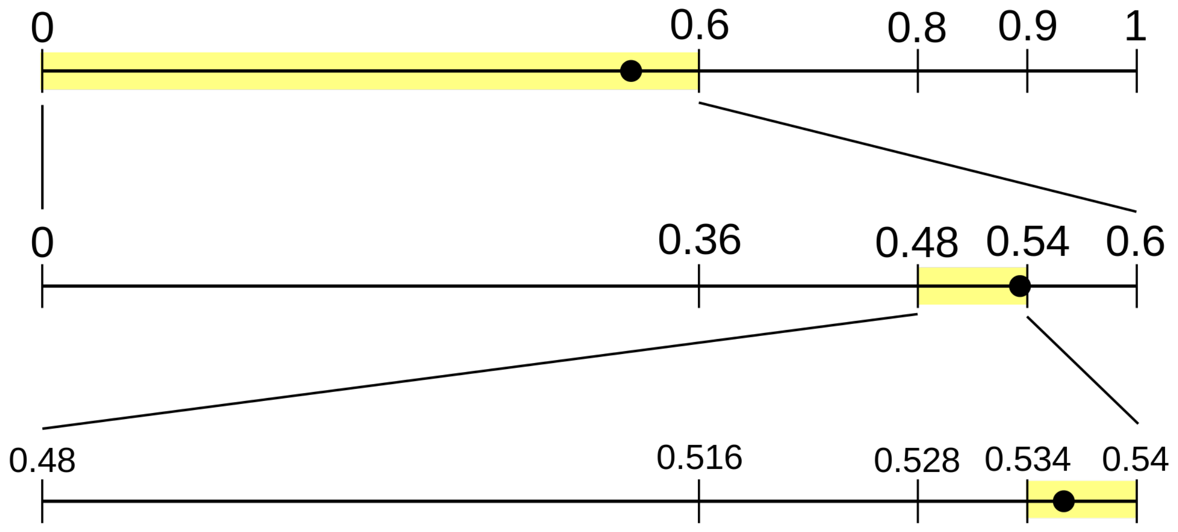


Figure 6: Arithmetic encoding example

As shown in the graphic above, the first iteration encodes the symbol 'A' and thus chooses the subdivision `[0, 0.6)`. Thus, low and high are set to the boundaries of the new division, 0 and 0.6 respectively. For the symbol 'C', we subdivide `[0, 0.6)` proportionally according to the frequencies and choose the third division, which is `0.48, 0.54`. For symbol `<EOF>`, we choose the fourth subdivision and end up with $(low, high) = (0.534, 0.54)$. Now we choose a value within low and high which will correspond to the encoded message that require the minimum bits to represent. We find that 0.1000101b corresponds to 0.53925, which is within our range. Since the value encoded will definitely be between 0 and 1, the leading '0.' can be removed. Thus, we encode the bits 1000101 which will become uniquely decodable into our original message.

2.5 OpenMP

OpenMP[22], or *Open Multi-Processing*, is an application programming interface in C, C++ and Fortran. Using a portable and scalable model, it allows programmers to easily implement multithreading via

```
int arr[100];

#pragma omp parallel for num_threads(10)
for (int i = 0; i < 100; i++) {
    arr[i] *= arr[i];
}
```

Figure 7: Example of C code using OpenMP

its flexible interface. During the preprocessing stage for C and C++, OpenMP can be linked via preprocessor directives. These directives indicate the section of the code to be run on a separate thread(s) from a common shared pool of threads to execute *independent* sections of the code in a parallelized manner. OpenMP is widely supported on most opreating systems and CPUs, which when combined with its ease of use, makes it a common choice for programmers meaning to exploit the multicore processors that can be currently found on standard machines.

The most common usage of OpenMP directives is the `omp parallel for` which splits up a `for` loop iterations between the threads where the loop body can be executed in parallel. Furthermore, the number of OpenMP threads used can be set via the addition of a `num_threads` clause in the preprocessor directive. As an example, each integer in a given array needs to be squared. Without the use of threads, a for loop can be used to iterate over all the elements, squaring them one by one. However, due to the fact that each iteration of the loop, i.e. the application of the squaring on each integer is independent, then OpenMP can be used to speedup the time spent by a maximum of the number of threads allocated.

2.6 Sequence Compression

The work done on the sequence compression algorithm described here is carried out by James Arram.[1]

2.6.1 Reference-based compression

Due to the high degree of similarities between the genomes across a multitude of species, reference-based compression has been utilized to drastically increase the compression ratios for DNA sequences. Of course, the first and foremost requirement is to come to a consensus on which reference sequence to employ as the base reference. After that, the compression algorithm merely needs to encode the differences between the given sequence and the reference sequence. Considering the fact that humans and bananas share 50% of their DNA, it is not difficult to imagine that in most scenarios the amount of information needed to be encoded would immediately be reduced 10-fold.

Now we need to find a way to represent the mappings between the reference sequence and the sequence being compressed. The most straightforward way is to use a 3-tuple <*pos, len, sym*> to do so. The first component, *pos*, represents the position at which the match begins. The second component, *len*, represents the length of the match. The third component, *sym*, represents the symbol after the match.

Sequence		**3-tuple**
Compress:	ATTATGTTCGATATAAT	
Reference:	AATTATGCTACGATCGATCGATCGATAAT	
Match 1:	ATTATG*T*	<2, 6, T>
Match 2:	TCGAT*A*	<14, 5, A>
Match 3:	ATAAT	<24, 5, _>

Table 3: Reference-based compression example

Now as we have these 3-tuples, it is necessary to find the number of bits required for each component. As the number of base pairs in the entire human genome is approximately 3 billion, 32-bits allows addressing of maximum of 4.29 billion, which is necessary and sufficient. Next, as each FASTQ read is no longer than 140 characters, using 8-bits enables us to encode a match length of 256 characters, which is greater than the maximum length of the read sequence. Finally, the number of characters allowed in the sequence needs to be determined. The DNA nucleotides (A, C, T, and G) and the RNA additional nucleotide (U) can appear in the sequence. Moreover, for matches that reach the end of the sequence, the next character will be set as NUL (ASCII 0). Furthermore, we have all the amino acid codes, non-determined nucleotide codes (such as N) and et cetera. To allow more compatibility, a full 8-bits corresponding to the ASCII value is encoded for the symbol.

2.6.2 FM-Index

Although the 3-tuples to encode the mapping between the sequence and the reference is defined above, the main challenge is to actually find these mappings. This stage is the most time-consuming of all the sequence compression processes. One of the most efficient ways to find the mappings is using the FM-index search operation.[4] The FM-index is a substring index based on the Burrows-Wheeler transform (hereafter referred to as BWT) and borrows its ideas of suffix arrays.

2.6.2.1 Burrows Wheeler Transform

The Burrows Wheeler Transform[3] is computed by initially appending the given string with a terminal symbol (‘\$’). Then, all possible rotations of the string (with the terminal symbol) is generated. The rotation is done by removing the first character of the string and appending it at the back, and thus for an input string of length l, there are $l+1$ rotations (as there is the terminal symbol). Afterwards, these rotations are sorted lexicographically. Note that the terminal symbol is treated as the character with the smallest value. Then, the order of the rotations after the sorting becomes the suffix array, while the last character of each of the rotations forms the Burrow Wheelers transform.

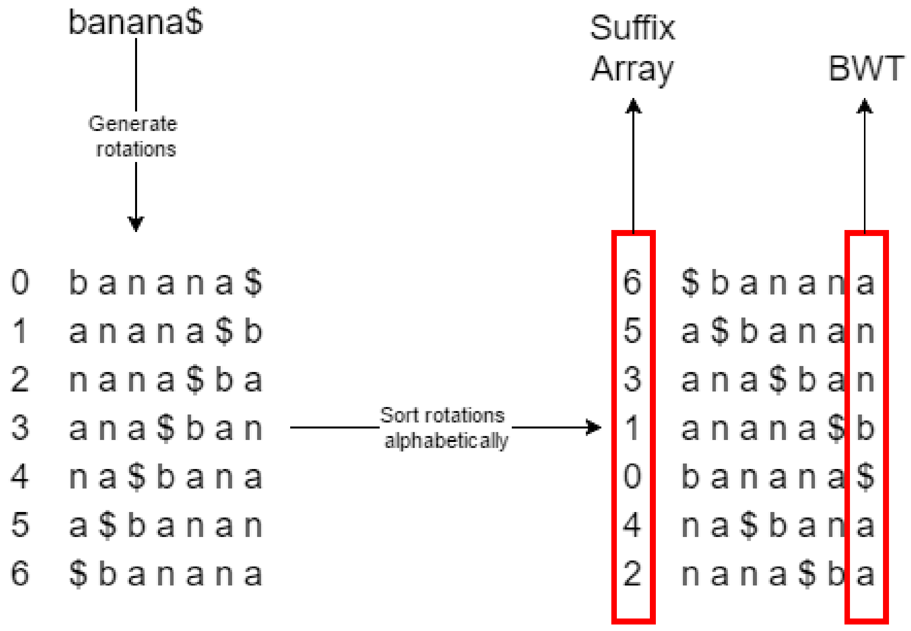


Figure 8: Burrows Wheeler Transform on string ”BANANA”

2.6.2.2 Computing the FM-index

After the BWT of the reference sequence is obtained, two tables are computed. Firstly, the table `C[c]` is a table of the total occurrences of lexically smaller characters in the input. Secondly, the table `Occ(c,k)` is a 2 dimensional table that stores the number of occurrences of a character c in the first k characters in the BWT. These two tables are critical to the operation and success of the FM-index as it allows substring searching in linear time with respect to the substring length, and is precomputed and stored when a sequence needs to be decompressed.

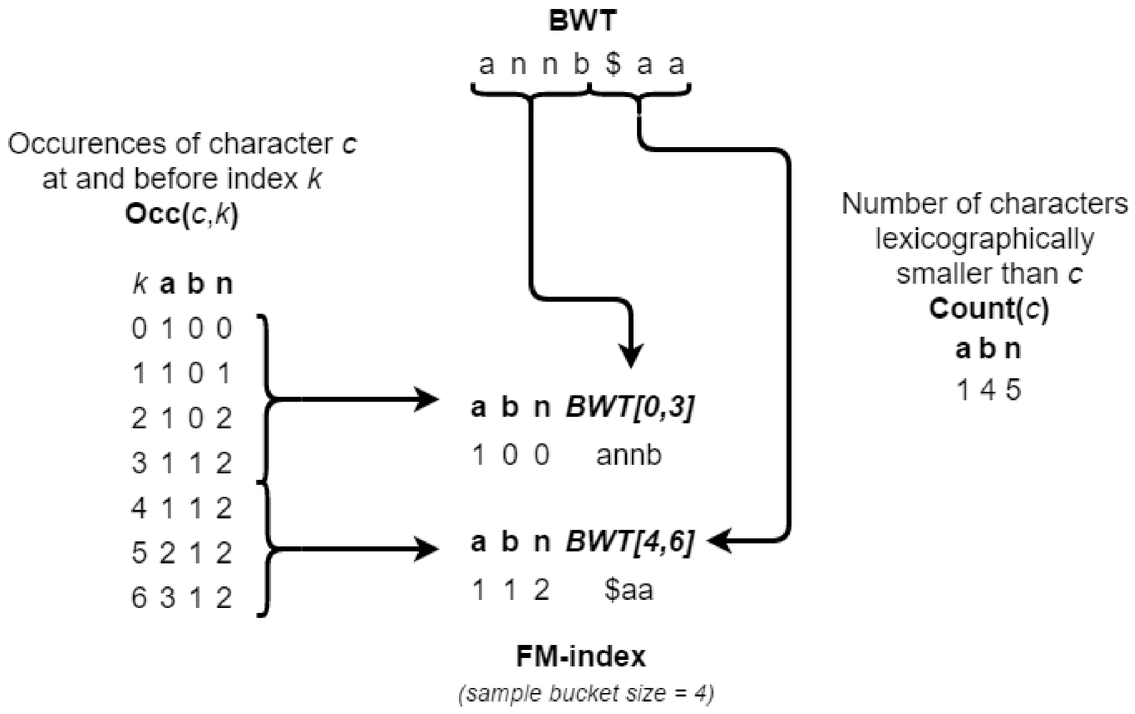


Figure 9: FM-index of string "BANANA"

As our reference sequence contains millions of base pairs, the `Occ(c,k)` table contains $\mathbb{B} \times |\text{Ref. Sequence}|$ values where $\mathbb{B}$ is the number of base pair characters, which is 4 ('A','C','G','T'), while $|Sequence|$ is the length of the reference sequence. Thus, the `Occ` table is put into buckets of size b. Then, only the `Occ(c,k)` values where k is a multiple of b are stored. To reconstruct the table, we store the section of the `BWT` corresponding to the deleted entries (i.e. `BWT[k+1, k+d)`) so the Burrow-Wheeler transform of the whole sequence does not need to be recomputed. By storing selective entries of the occurence table, it effectively reduces the size of the precomputed table b-fold. At the beginning of the compression algorithm, the `Occ` table is reconstructed using the stored values `Occ(c,k)` where $\exists n \in \mathbb{N} \rightarrow k = nd$ and finding the occurrences of each character in `BWT[k+1, k+d)`.

The advantage of FM-index over other sequence mapping approaches such as hash tables are that the mapping performance is independent of the reference sequence length and thus the compression speed is not reduced even when using a huge sequence such as the full Human genome with 3 billion base pairs as the reference sequence. Furthermore, since genomes are rarely updated the FM-index would only be computed once for all the compression jobs. There will not be any additional overhead such as hash collision handling when using hash tables.

2.6.2.3 Finding the matches

Now that the FM-index has been precomputed, the FM-index search algorithm described below will compute the matches between a given reference string and a substring.

Now that the matches can be found, the compression of a sequence is simplified. The proceduer **FM-Search** returns the indexes of the match of longest possible length. Then, as mentioned previously, the tuple *<pos, len, sym>* is encoded. If after the tuple is added, there are still unmatched symbols, then **FMSearch** is called starting from the symbol *after* the first mismatched symbol since that first mismatched symbols is encoded in the tuple.

Algorithm 1 FM-index search

Require: FM-index *FM*, reference string *R*, substring *S*, suffix array *SA*, bucket size *d*
Ensure: an array *p* of position indexes where *S* occurs in *R*

```
 1: procedure GetOcc(b, c, i)                    ▷ Computes Occ(c,i) given FM-index bucket entry b
 2:     $count \leftarrow$ b.marker[c]
 3:     for $j \leftarrow 0$ to $i \mod d$ do
 4:         if $c =$ b.BWT[j] then
 5:             $count \leftarrow count + 1$
 6: return $count$

 7: procedure FMSearch(FM, R, S, SA, d)
 8:     $low \leftarrow 0$
 9:     $high \leftarrow |R|$
10:     for $i \leftarrow |S| - 1$ to 0 do
11:         $low \leftarrow GetOcc(FM[low - 1/d], S[i], low - 1)$
12:         $high \leftarrow GetOcc(FM[high/d], S[i], high)$
13:         if $low \geq high$ then                    ▷ Substring does not occur in reference
14:             break
15:     $p \leftarrow$ empty array
16:     for $i \leftarrow low$ to $high - 1$ do
17:         append SA[i] to $p$
18: return $p$
```

2.6.3 Decompression

As in the FMSearch algorithm, the string to be matched is traversed from the right to the left. Thus the order of the tuples in the decompression stage would need to be similarly traversed from right to left. The decompression is much simpler than the compression stage as each tuple's decompressed value is simply the substring of the reference sequence starting at the encoded offset and of the encoded length, then followed by the encoded symbol.

2.6.4 Reverse complements

Above we have outlined the two major steps of the compression algorithm for sequences - computing the FM-index for the reference sequence and matching it to the input sequence to be compressed. Now further modifications that improve the overall compression ratio will be discussed.

In next generation sequencing technology, the individual DNA fragments are amplified after they are hybridized into the flowcells. This indicates that both strands of the DNA will be read and sequenced. For example, the sequenced human genome `ERR161544` has 74 million reads of 100 base pairs, which yields double the amount of base pairs in a single strand of human DNA (approximately 3 billion). If the read begins at the same end for both strands, then the combined reads would be two different sequences. Thus, sequencing technologies orient the strands so that all the reads begin at the 5' end (phosphate group) and end at the 3' end (hydroxyl group).

Then, the relationship between the reads of the two strands are that they are reverse complements of each other. Since the reference genome is one strand, half of the reads that are from the other strand would yield a much higher compression ratio if its reverse complement was encoded instead. However, since it is not known whether a particular read is from the same strand as the reference or not, to get the best compression rates, for each read, both the original read sequence and the reverse complements are mapped to the reference. The number of tuples in the mapping is then compared, and the set with the fewer tuples is encoded while the set with the greater amount of tuples is discarded. However, to decompress the information, whether the reverse complement was utilized or not would need to be encoded as well. As it there are only two possibilities for the boolean state of whether the reverse complement was encoded or not, only one bit is necessary to encode the information per FASTQ entry.

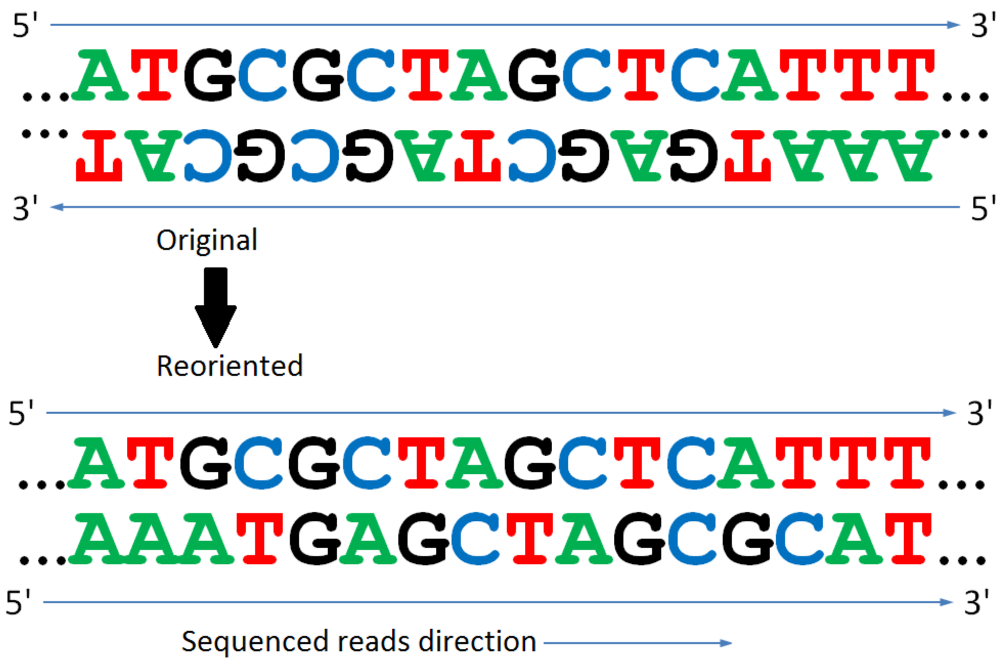


Figure 10: Orientation of sequencing reads

During the decompression stage, after finding the corresponding values of the tuples corresponding to a particular read, then the bit denoting whether the reverse complement is used would need to be checked. If the reverse complement is indeed used, then the reverse complement of the decoded sequence is output to the decompressed file. This stems from the fact that the reverse complement function is the composite of two involutions (i.e. it is the inverse of itself), namely *reverse* and *complement*. The composite of two involutions is itself an involution, and thus applying the reverse complement function the second time will revert the sequence into its original:

$$rc(seq) = rc^{-1}(seq) \Rightarrow rc(rc(seq)) = seq$$

2.6.5 Mapping unknown base pairs

One of the major assumptions in the referential mapping algorithm detailed above is that the reference and the input sequence consists of the same characters. More specifically, that the input sequence cannot contain characters that are not in the reference sequence - as it would be impossible to obtain a mapping and that since our stored FM-index only contains the occurences for symbols that actually occur, having characters not in the index would lead to undetermined behaviour. Although most sequences from the newer sequencing machines have the heightened ability to differentiate between the different nucleobase signals during the sequencing process, some of the older sequenced data contain many low quality reads plagued with unknown bases.

Thus, it is necessary to process the reads before invoking the mapping algorithm. The proposed method is thus; firstly, the read sequence is split into contiguous sections of either ACTG bases or unknown (non-ACTG, usually denoted by `N`) characters. Then, the substrings containing the ACTG bases are mapped as normal using the matching algorithms above. The non-ACTG substrings are then resplit into contiguous sections of the same character. Then, the tuple of $<$*0, len, c*$>$ is encoded, where *len* is the number of repeated non-ACTG character, and *c* is the non-ACTG character itself.

2.6.6 Selective Decompression

Now that the information required to be encoded has been determined, it is possible to find a way to enable selective decompression. The easiest way to do this is by counting and storing the number of

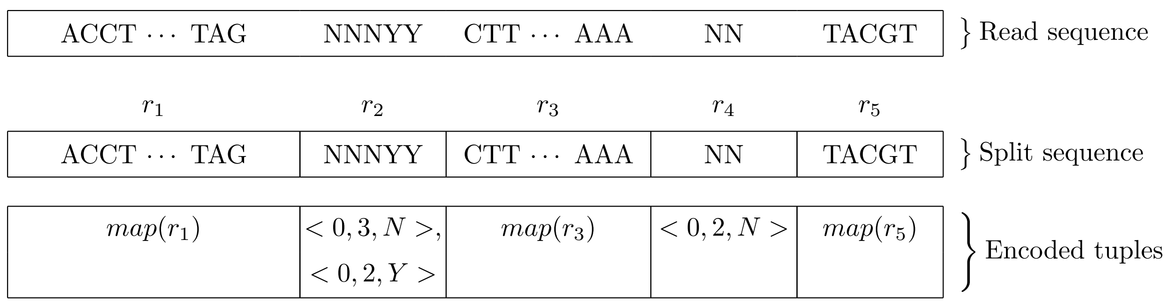

ACCT ⋯ TAG NNNYY CTT ⋯ AAA NN TACGT
Read sequence
r_1 r_2 r_3 r_4 r_5
ACCT ⋯ TAG NNNYY CTT ⋯ AAA NN TACGT
Split sequence
$map(r_1)$ $< 0, 3, N >$, $< 0, 2, Y >$ $map(r_3)$ $< 0, 2, N >$ $map(r_5)$
Encoded tuples

Figure 11: Encoding structure of Quality Scores

<*pos, len, sym*> 3-tuples for each FASTQ entry. Each of these counts are encoded using 8-bits in a separate file. Now suppose the m-th FASTQ entry is wanted. The first m bytes of the file containing the counts are read and then summed. Once we find the sum of the total amount of tuples utilized to encode the first m - 1 entries, the compressed tuples file is seeked to an offset of $\sum_i^{m-1} count_i \cdot$ (bits per tuple) from the start of the file. Then, as we know the number of tuples corresponding to the m-th entry, the sequence can be reconstructed easily.

2.7 Maxeler Dataflow Computing Engine

Dataflow computing[20] is an innovative approach to high performance computing software. Unlike most modern CPUs, which can be described as versatile workers that handle what its instruction set allows, in dataflow computing the computation is done by thousands of tiny cores where each can handle one instruction. The connection between the cores are determined via prior configuration. Dataflow computing exploits massive parallelism as streams of inputs through these interconnected cores enable extremely high *throughput* which is evident by the streamed outputs, similar to the production at a factory line. James Arram has implemented the sequence compression using the Maxeler Dataflow Engine, which drastically cuts down the time utilized for the compression of the sequence.

2.8 Compression Utilities

To evaluate **compress**, the compression tool described in this report, six other compression tools have been chosen. Two of them are popular general purpose compression tools. The other four are specifically targeted towards the compression of FASTQ data.

General purpose compression tools

2.8.1 pigz

pigz, which stands for parallel implementation of gzip, is a compression algorithm that is based on the **gzip** algorithm but modified to utilize multiple cores and processors. The input is broken up into 128 KB chunks and uses the DEFLATE algorithm to compress the chunks. These compressed values are then concatenated and wrapped inside a header and trailer that encodes the metadata about the compressed file such as check values to guarantee integrity.

The input blocks will have a portion of the previous block used as the initial dictionary. Thus, the decompression cannot be parallelized. However, the decompression will create three other threads for reading, writing and check calculation.

2.8.2 pbzip2

pbzip2 is the parallel implementation of the bzip2 file compressor. It utilizes a series of compression techniques applied one after another, namely

1. Run-length encoding (RLE) of initial data
2. Burrows–Wheeler transform (BWT) or block sorting
3. Move to front (MTF) transform
4. Run-length encoding (RLE) of MTF result
5. Huffman coding
6. Selection between multiple Huffman tables
7. Unary base 1 encoding of Huffman table selection
8. Delta encoding of Huffman code bit-lengths
9. Sparse bit array showing which symbols are used

in that order. The decompression will apply the decompression of each of the above techniques in reverse order.

Domain-specific genomic compression tools

2.8.3 fqzcomp

fastqzcomp[2] is one of the entries in Pistoia Alliance's SequenceSqueeze contest for the compression of FASTQ files. The compression of identifiers utilize tokenization and then delta encoding. The quality scores attempt to model the correlations between successive quality scores using preset rules. The sequence encoding uses the previous *b* base pairs to predict the next base pair, where b can be configured by the user.

2.8.4 LWFQZip

LWFQZip[32] is a FASTQ/FASTA compression tool. It uses incremental coding to encode the metadata. For the sequences, a referential compression is used. Thirdly, the quality scores undergo run-length encoding. Finally, all of the components are compressed at the end with the Lempel-Ziv-Markov chain algorithm (LZMA).

2.8.5 fastqz

fastqz[2] uses a modified incremental encoding algorithm for the metadata compression. The quality scores are compressed by using byte codes to replace common runs and common groups of size 2 or 3 in the quality scores. The sequence compression has two options - either a reference based approach or a derivative of entropy encoding that encodes common groups of base pairs in fewer bits. However, it should be noted that the *compression is not completely lossless* as it assumes that if there is a code that is not A, T, C or G, then the quality score is 0. However, this can be shown to be not the case with a cursory examination of the FASTQ files.

2.8.6 quip

Quip[16] extends the reference-based compression and uses a probabilistic prediction model using *De novo transcriptome assembly.* Instead of using an external reference, **quip** can generate messenger RNA via sampling the sequenced reads. The metadata is tokenized and then encoded using delta encoding. The quality scores are modelled using markov chains with order 3.

2.9 Summary

In this chapter we have presented a wide range of background information. Starting from a bioinformatics perspective and exploring the motivation of this project, we then move on to an introduction to the FASTQ and FASTQ genomic data formats that our compression tool targets. A multitude of data compression techniques and algorithms are covered accompanied by examples and analysis of scenarios that the algorithms are optimal in, namely: (1) run-length coding suitable for frequent consecutive repetitions of the same character, (2) Huffman coding which generates a variable length prefix code by assigning symbols with higher frequency shorter codewords and vice versa, (3) arithmetic coding which encodes the entire given message into a single value, (4) the Burrows Wheeler Transform which takes an input string and transforms it into a permutation of the original string that contains runs of similar characters, and (5) the FM-index, which is an effective substring match algorithm based on the Burrows Wheeler Transform. Moreover, previous work regarding effective sequence compression is described. Software and hardware acceleration techniques is then introduced to the read. Finally, a brief summary of various compression utilities that are currently used for the purpose of FASTQ and FASTA compression is given.

3 Metadata/Sequence Identifiers

In this chapter, the following contributions are presented:

- Division of the FASTQ and FAST metadata into separate segments with common characteristics;
- Development of encoding algorithms for different types of segments;
- Exploration of methods that enable selective decompression; and
- Designing an overall encoding structure that allows lossless decompression and a certain degree of backwards compatibility.

3.1 Overview

As mentioned previously, the first line of each entry in a FASTQ (and FASTA) file contains the metadata/sequence identifier for a particular read. This line begins with the symbol '@' followed by an assortment of different values representing optional information about the read such as the instrument name, flow cell identifier, coordinates, length and et cetera.

One of the major problems with devising a suitable compression algorithm is that there is not a single standard for the format of the metadata. The format is not even consistent between different sequencing machines *from the same manufacturer.* Take for example, a metadata from an early version of Illumina machines would have the format of:

@HWUSI-EAS100R:6:73:941:1973	
HWUSI-EAS100R	the unique instrument name
6	flowcell lane
73	tile number within the flowcell lane
941	'x'-coordinate of the cluster within the tile
1973	'y'-coordinate of the cluster within the tile

Table 4: Example of Metadata from Illumina sequencing machines prior to version 1.8

In Illumina Casava 1.8 the sequence identifiers underwent massive changes to include much more informaton, and was vastly different from the previous example:

@EAS139:136:FC706VJ:2:2104:15343:197393 1:Y:18:ATCACG	
EAS139	the unique instrument name
136	the run id
FC706VJ	the flowcell id
2	flowcell lane
2104	tile number within the flowcell lane
15343	'x'-coordinate of the cluster within the tile
197393	'y'-coordinate of the cluster within the tile
1	the member of a pair, 1 or 2 (paired-end or mate-pair reads only)
Y	Y if the read is filtered, N otherwise
18	0 when none of the control bits are on, otherwise it is an even number
ATCACG	index sequence

Table 5: Example of Metadata from Illumina Casava 1.8

In order to accommodate the compression of data from a multitude of sequencing platforms, it is essential to dissect the sequence identifiers and find the common denominators in order not to restrict the compression utility in its scope.

After delimiting the sequence identifiers into individual fields, it is easy to identify four major **types** of values that are being encoded, namely *Constant Alphanumeric, Alphanumeric, Numeric* and *Autoincrementing Numeric.* Furthermore, it can be observed that the type and the order of the fields in each sequence identifier of the same file is constant. Using this information, we can exploit the similarities to find the optimal encoding for each type of values.

3.2 FASTQ Sequence Identifiers

3.2.1 Fields

In order to decode the various fields in the sequence identifiers, the program must firstly know the number of fields. We encode this number in 8 bits, thus supporting a maximum of 256 fields in the sequence identifiers (which should be overly sufficient). This information is necessary to start the decoding process, and thus will be at the very beginning of the encoded file.

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
No. of Fields	Reserved

Now the next 24 bits are reserved to represent other information detailed later on.

3.2.1.1 Types of Fields

By looking through various sequence identifier formats, we can first separate the fields into two major categories - namely numeric and alphanumeric. Numeric fields only contain integers ([**0-9**]) while alphanumeric fields can contain other ASCII/UTF-8 characters.

The alphanumeric fields can be further subdivided into constant and non-constant fields. Constant fields are utilized to denote instrument ids while non-constant fields represent various pieces of information ranging from flowcell ids to index sequences. Moreover, one major archetype of numeric fields are auto-incrementing fields that denote the run id. Now the four identified common types fo field can be encoded in 2 bits, and the encoding is explicitly defined in table 6.

Field type	**Bit encoding**
Const. Alphanumeric	0b00 [2]
Alphanumeric	0b01
Numeric	0b10
Auto-incrementing Numeric	0b11

Table 6: Fields Encoding

3.2.1.2 Width of Fields

In order to achieve maximum compression, we can reduce the number of bits used to encode each field by finding the minimum number of bits that can losslessly store the fields values by being able to accommodate the maximum value (for numeric fields) or maximum length (for alphanumeric fields) that occurs in the file.

For example, if we know that the ranges of the *x and y tile coordinates* values to be within $0 \leq x, y \leq 1000000 < 2^{20}$, we can use 20 bits to store the values (instead of using the usual 32 bits width common to integer types). For the alphanumeric fields, if we know the maximum length of the values, we can also set the amount of bits required to encode the values without any loss of information. Now we use 14 bits to denote the number of bits to be allocated towards each field (which allows each field's value to be encoded in a maximum of $2^{12} = 4096 bits = 512 bytes$, which would allow the encoding of 585 characters (using the assumption made earlier about the domain of the valid alphanumeric characters which allows 7-bits encoding per character) for alphanumeric fields and an unsigned integer value of 2^{4096} (which *should* be sufficient).

[2]The prefix *0b* will be used to denote that the following value is binary. Similarly, the prefix *0x* will be used to denote a hexademical notation.

3.2.1.3 Determining the Type of Field

Now that the archetypes of common fields and their values have been defined, it is crucial to find an algorithmic approach to determining the type of field given an array of FASTQ reads. The pseudocode below outlines how each type of field is identified.

Algorithm 2 Determining the type of fields in the metadata/sequence identifier

Input: an array of FASTQ reads
Output: an array `field_type` containing enumerated values corresponding to the field type

```
1: procedure AnalyzeFields(reads)
2:     field_count ← number of delimiters in first metadata + 1
3:     sets[field_count] ← initialize field_count empty sets
4:     for r in reads do
5:         tokens[field_count] ← split r using delimiters
6:         for i:=1 to field_count do
7:             append tokens[i] to sets[i]
8:     for i:=1 to field_count do
9:         if $set[i].size = 1$ then
10:            $field\_type[i]$ ← Constant alphanumeric
11:        else
12:            numeric ← true
13:            for t in set[i] do
14:                if isnumeric(t) then
15:                    numeric ← false
16:                    break
17:            if numeric then
18:                if $max$(set[i]) = reads.size and $min$(set[i]) = 1 and set[i].size = reads.size then
19:                    $field\_type[i]$ ← Auto incrementing
20:                else
21:                    $field\_type[i]$ ← Numeric
22:            else
23:                $field\_type[i]$ ← Alphanumeric
24: return field_type
```

3.2.1.4 Encoding the information about the various fields

Now we have the necessary information to develop an encoding to represent the metadata about the fields. However, there are two things to note: firstly, the width does not apply to the auto-incrementing numeric field as data is not encoded. However, to combat the possibility of the auto-incrementing field not starting from 1, we allocate 30 bits to denote the initial value of the field. Moreover, for the constant alphanumerical fields, we immediately encode the value with the number of bits specified in the width parameter.

So for the example Illumina Casava 1.8 sequence identifier above, the metadata about the compressed identifier will be:

From the metadata, more specifically the widths, it is stated that each compressed entry will be $42 + 4 + 15 + 20 + 20 + 1 + 6 + 6 = 114 bits = 14.25 bytes$ by summing up the widths of the non-constant alphanumeric fields and the numeric fields. Contrasted with the original 53 characters or 53 bytes due to the ASCII encoding, the compression ratio per entry would be **26.9%**. Although there is the additional overhead due to the field metadata at the beginning of the file, this becomes insignificant after hundreds of FASTQ entries - and FASTQ files can easily reach up to *hundreds of millions* of entries.

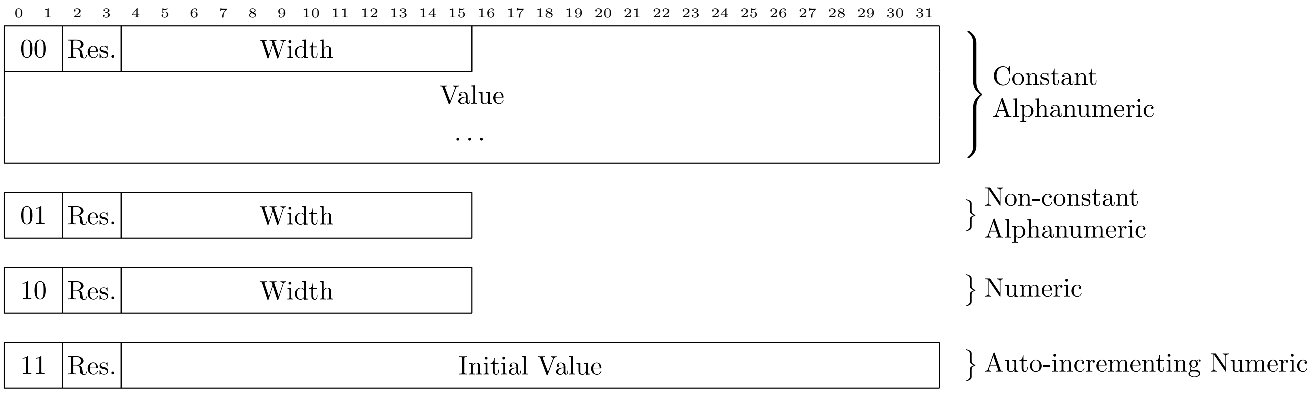


Figure 12: Encoding structure of Field Metadata

Value	Field description	Type	Maximum size	Width	Value
EAS139	the unique instrument name	0b00	6 * 6 bits = 36 bits	0b100100	0x3DAB35DFD
136	the run id	0b11	*auto-incrementing*		0x1
FC706VJ	the flowcell id	0b01	7 * 6 bits = 42 bits	0b101010	
2	flowcell lane	0b10	16 = 4 bits	0b100	
2104	tile number within flowcell lane	0b10	20000 = 15 bits	0b1111	
15343	'x'-coordinate of the cluster	0b10	20 bits	0b10100	
197393	'y'-coordinate of the cluster	0b10	20 bits	0b10100	
1	the member of a pair	0b10	1 bit	0b1	
Y	Y if the filtered read, N otherwise	0b01	1 * 6 bits = 6 bits	0b110	
18	control bits	0b10	64 = 6 bits	0b110	
ATCACG	index sequence	0b00	6 * 6 bits = 36 bits	0b100100	0xD5BCD720

Table 7: Field Metadata Encoding of Illumina Casava 1.8 Sequence Identifier Example

3.2.2 Delimiters/Separators

As mentioned above, in order to provide a lossless compression algorithm, every bit of information must be saved. This includes the separators utilized to delimit the various fields of the sequence identifiers.

We can make a few observations about the separators. Firstly, the first character of the sequence identifier will be the '@' symbol, which is intended to indicate the start of each entry in the FASTQ file. As this is constant throughout **all** FASTQ files, the '@' symbol does not need to be encoded. Secondly, the separators and their respective orders within the sequence identifiers are preserved throughout the entries within the same FASTQ file. Although these two observations may not be 100% correct as the FASTQ format utilized does not have a singular standard, from various FASTQ files outputted from a multitude of hardware seem to agree on these observations and thus we will assume these assumptions hold in order to simplify the encoding process.

Now we will attempt to find the minimum amount of bits required to form a bijection between the separator symbols and the values. From the MAQ tool we find that the possible allowed separators are **space (' '), period ('.'), colon (':'), hyphen ('-') and hash ('#')**. Due to the five possible values, we require 3 bits to represent each possible value. Furthermore, the user can define delimiters by passing it as an argument to the program. We use the bits `111` in the separator metadata to denote that this is a user-defined separator. If the `111` is read, then the following 8-bits correspond to the UTF-8 value of the user-defined delimiter.

Delimiter	Bit encoding
Space (' ')	000
Period ('.')	001
Colon (':')	010
Hyphen ('-')	011
Hash ('#')	100
User-defined	111

Table 8: FASTQ Separator Encoding

We store the separators, in their respective orders, immediately after where the last of the field metadata has been stored. Now we have the overall structure for the metadata regarding the fields and separators utilized in the original FASTQ file.

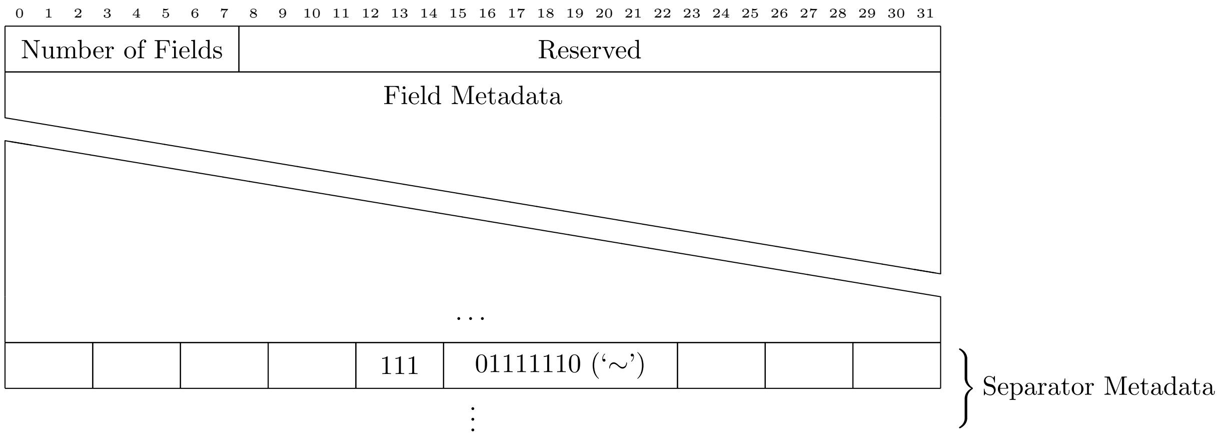


Figure 13: Encoding structure of Compression Metadata

3.2.3 Further Compression of Field Values with Selective Decompression

Currently, the basic encoding structure for the metadata has been defined above where non-constant values are merely transcribed and encoded by removing *redundant* bits, further savings on the output compressed filesize can be achieved through the utilization of various methods that focuses on eliminating *repetition*. However, in order for the decompression process to know whether selective decompression has been enabled or not, the user's selection during the encoding process needs to be encoded. The fourth bit of the reserved section in the header will be set if selective decompression is enabled (and thus restricting the choice of algorithms for further compression of field values).

3.2.3.1 Restricting domain of alphanumeric fields

If we limit the domain of alphanumeric fields to the lower and upper case English alphabet in addition to the numbers and the underscore symbol (**[a-zA-Z0-9_]**), we can utilize 6 bits to encode each character instead of the 8 bits common to character encodings (UTF-8 and ASCII), thus achieving a further 25% improvement in compression rates for alphanumeric fields. Now, if all the metadata entries are determined to only contain the alphanumeric characters and no more than two other characters excluding the characters treated as delimiters, then this mapping from ASCII or UTF-8 values into the 6-bits can be utilized. If this is indeed the case, then the first bit of the reserved section will be set.

3.2.3.2 Mapping alphanumeric fields

Many alphanumeric fields can only take a limited set of values within the same file. For example, the flowcell id and the filtered read from Table 5 only takes up very few values compared to the maximum we have allowed for (every permutation of every alphanumeric character). Thus, if we identify that there is very little variance in the possible values, we can simply store the map of the available values and encode the index of the value within each entry. In the above example, we can save 5 bits per entry for the filtered read field, and assuming that there are less than 16000 flowcell ids, we can store that in 14 bits which would be a saving of 26 bits per entry. This is a substantial decrease of approximately 30% in the amount of bits to encode each entry for the example above.

To enable this, we set the first reserved bit in the non-constant alphanumeric field. The width section will now be utilized to represent the number of entries for the map, so a maximum of 4096 entries for the map is allowed. Immediately following this field metadata, the alphanumeric values will be copied and then null terminated. There should be exactly the amount of null terminated strings as stated in the prior metadata. The order of the values will correspond to their key mapping (using 0-indexing). The width of the encoded value for each FASTQ entry will then be the minimum number of bits required to represent the number of mappings, which can be inferred without being encoded in the metadata.

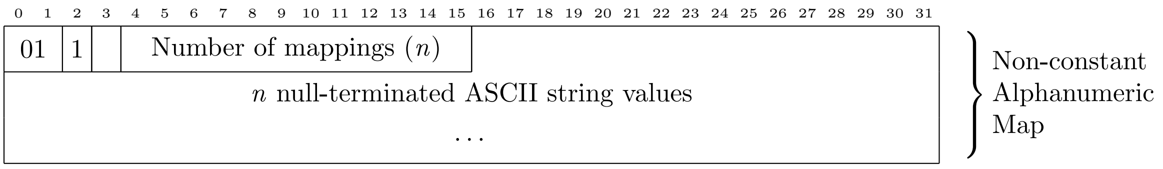


Figure 14: Metadata for Alphanumeric Map

Moreover, we can do this for numeric fields as well. This would increase the compression rates if the range of the numeric fields is much larger than the number of distinct values for the field. However, we will still retain the width for the numeric field in the metadata as there is no null termination for integers.

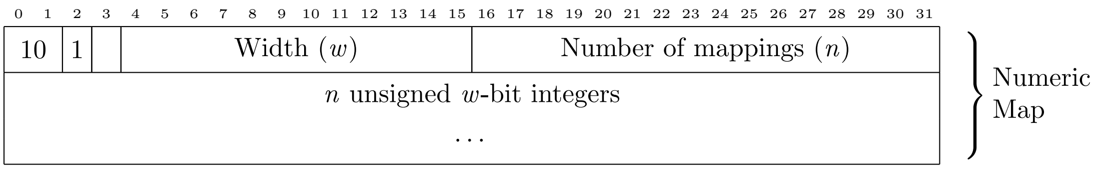


Figure 15: Metadata for Numeric Map

So the subsequent block of $n \cdot w$ bits will correspond to the mapped values, and the encoded value will once again correspond to the order of the integer values in the map and the width of the encoded value in the entries will be the minimum number of bits required to represent the number of mappings.

Compression Rate of first 10 million entries of ERR161544

Field	Metadata / bits	Entry / bits	Total / bits
1. **Original**			
• ASCII	0	384 to 464	4311111120
2. **Compressed**			
• Constant Alphanumeric (9 ASCII)	79	0	79
• Autoincrement	32	0	32
• Constant Alphanumeric (10 ASCII)	86	0	86
• Constant Alphanumeric (1 ASCII)	23	0	23
• Numeric (Max: 5)	16	3	30000016
• Numeric (Max: 21361)	16	15	150000016
• Numeric (Max: 200794)	16	18	180000016
• Alphanumeric Map (5 10-char values)	416	3	30000416
Total	684	39	390000684

Then we can now benchmark the algorithm and compare it to the original files and other compression algorithms.

3.2.4 Further Compression of Field Values without Selective Decompression

Without the need for selective decompression, we can employ other methods in addition to those listed for when selective decompression was enabled.

3.2.4.1 Incremental encoding of numeric fields

For numeric fields, the size of the fields may be decreased if we first scan through the file to check for the differences between successive fields. Then instead of encoding the numeric value, it is possible to encode the difference between the values instead. This will be guaranteed to be no worse than directly encoding the value as the maximum value for the difference encoded is less or equal to than the maximum value itself. If it is observed that there are no huge jumps between successive entries, then saving 3-4 bits per entry would entail an overall decrease in the magnitude of megabytes.

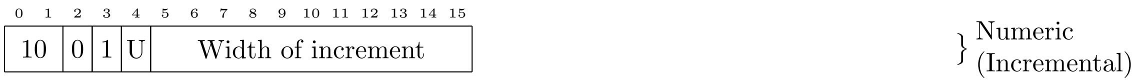


Figure 16: Metadata for Numeric (Incremental)

In order to denote this, we set the 2nd reserved bit in the metadata. Then the next bit denotes whether the increments are signed (set U to 0) or unsigned (set U to 1). The remaining 11 bits denotes the width of the encoded increment values in each entry.

3.2.4.2 DEFLATE for non-mapped alphanumeric fields

For non-constant alphanumeric fields that can take a large amount of values and thus the aforementioned mapping is not performed, we can utilize the DEFLATE algorithm to encode similarities between the current value and previous values.

DEFLATE matches sections of strings with previously encoded sections. For example, in the string "ABCABCABC", we can encode the literal 'ABC'.

```
ABCABCABC                   - original string
ABCABCABC                   - current section
   ^^^^^^
ABCABC                      - current section offset by 3
ABC[offset = 3, length = 6] - compression
```

Then we encounter the string 'ABC' once more, which we can match with the previously encoded 'ABC' with an offset of 3 (the difference between the start of the previous string and the current string) and the match is of length 3. However, we can find that that 'ABC' is repeated once more, and so that the string beginning from index 0 of length 6 matches the string beginning from index 3 of length 6! Thus we can actually put down that the match is of index 3 and length 6.

After this substring matching, we encode the offset/length pairs using Huffman encoding.

In the best case scenario, this produces improved compression rates if the entries have repeating substrings as seen above. However, we often have to discuss the worst case asymptotic complexity, which in this case is still O(n) as we just have the whole input copied as a literal and the overhead of headers and et cetera is insignificant compared to the values.

3.2.4.3 Huffman encoding of mapped alphanumeric fields

If an alphanumeric field takes a *very* limited amount of values and the distribution of the alphanumeric field value occurrences is highly uneven, Huffman encoding can prove to be useful in reducing the number of encoded bits. The distribution of the occurrences of the various values needs to be sufficiently uneven, or else the average number of bits required to encode each alphanumeric field entry would simply be equal to the *integer to string* mapping described in section 3.2.3.2. Obviously, this would yield higher compression time as the metadata analysis stage not only requires keeping track of the number of unique values for alphanumeric fields but the frequencies of the values as well. Moreover, the Huffman code will need to be constructed, and then serialized. The algorithm of the serialization can be described recursively.

If Huffman encoding is enabled, the alphanumeric field metadata's 2nd reserved bit would be set. Afterwards, the width field will be removed as the Huffman code is variable-length, and the bit encodings of each alphanumeric value can be determined from the Huffman tree itself.

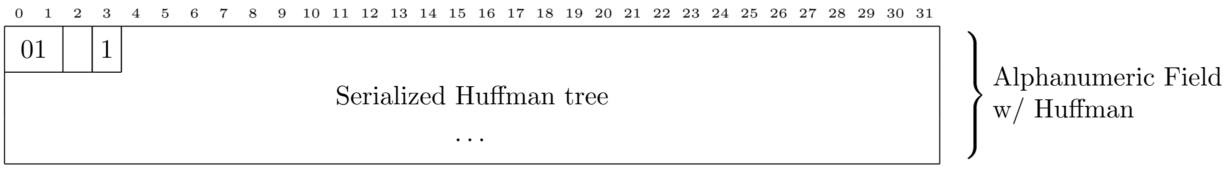


Figure 17: Metadata for Numeric Map

Algorithm 3 Serialization of Huffman Tree

Input: Root node of tree *node*, buffer to serialize to *buffer*
Output: Serialized Huffman tree at *buffer*

```
1: procedure SERIALIZE(node, buffer)
2:     if $node$ is leaf node then
3:         $buffer.write(0b1)$
4:         $buffer.write(node.value)$            ▷ Null-terminated string value in UTF-8
5:     else                                      ▷ Recursively serialize child nodes
6:         $buffer.write(0b0)$
7:         serialize($node.left$)
8:         serialize($node.right$)
```

3.3 FASTA Sequence Identifiers

3.3.1 Fields

Similarly to the FASTQ format, the first line of each entry in a FASTA file contains the sequence identifiers.

```
>gi|51466650|ref|NT_007914.14|Hs7_8071 Homo sapiens chromosome 7 genomic contig,
reference assembly
```

From FASTA files we can observe that the fields are very much similar to the FASTQ files, and thus we will encode them in a similar fashion as described in 3.3.1.

However, there is a potential problem - that the last field in the FASTA sequence identifier, which is a short word description of the sequence (*Homo sapiens chromosome 7 genomic contig, reference assembly* in the example above), may , may not fit in the 32 bytes maximum limit imposed in the FASTQ encoding. Thus we expand the maximum width of the alphanumerical fields (both constant and non-constant) to 128 bytes by allocating an additional 2 bits for the width encoding for the metadata (to a total of 8 bits).

3.3.2 Separators

From the example of a FASTA sequence identifier above, it is shown that the separators utilized for delimitation purposes differ from those used in FASTQ formats. For example, the last field in the FASTA sequence identifier, which is a short word description of the sequence (*'Homo sapiens chromosome 7 genomic contig, reference assembly'* in the example above), is one single field that contains the space character.

Looking through various FASTA files, we can find that there are five characters used as separators: **bar ('|'), period ('.'), underscore ('_'), hyphen ('-'), and space (' ')**. Furthermore, similarly to FASTQ, we allow an additional symbol to denote user-defined delimiters similar to that of FASTQ above. With 6 possible separator symbols, 3 bits are utilized to encode the separators. The encoding is specified in table 9.

Field type	Bit encoding
Bar ('|')	000
Period ('.')	001
Underscore ('_')	010
Hyphen ('-')	011
Space (' ')	100
User-defined	111

Table 9: FASTA Separator Encoding

Secondly, each line starts with the '>' symbol instead of the '@' symbol in FASTQ files. This does not to be encoded as it is consistent throughout the FASTA files.

Although it can be deduced from the file extensions of the original file and compressed file whether it is a FASTQ or FASTA formatted file, it would be useful to further guarantee the proper decompression if one bit was utilized to denote whether it is a FASTA metadata (and thus FASTA file). If the input file is of the FASTA format, the 12th bit of the header is set.

3.4 Deleted entries and Auto-incrementing numeric fields

When the FASTQ sequence data is analyzed, it is possible that some data entries that are not significant may be removed. This will cause the decoding algorithm to incorrectly find the values of the corresponding run-ids and other auto-incrementing numeric fields. To tackle this problem, we add an optional flag to be passed to the program when encoding. Whether this flag is set will be stored in the 3rd bit of the **Reserved** section of the metadata (and the 10th bit overall).

Now we need to attempt to encode the information regarding the skips in an auto-incrementing numeric field. One of the most obvious ways to achieve this is When this flag is set, then for every entry, instead of not encoding any information for the autoincrementing fields, 1 bit will be assigned to denote whether this value is incremented as usual (**'0'** for increment, and **'1'** otherwise). If a '1' is read, then the next 32 bits will represent the value of the field. If we further assume that the values for the field is strictly increasing, we can cut the allocated 32 bits to 31 and encode the difference between the current value and the previous value instead. Of course, if the difference cannot be stored within 31 bits, then the FASTQ file would be required to be split into two files, or split into separate batches (see section 6.1).

Sequence Identifier	Auto-increment Bit	Auto-increment Value
@EAS139:136	0b0	
@EAS139:137	0b0	
@EAS139:200	0b1	0x003f
@EAS139:201	0b0	

Table 10: Skipped Auto-incrementing Field Example

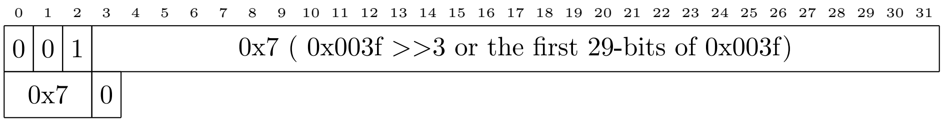


Figure 18: Encoding skipped auto-incrementing fields

Nonetheless, the above approaches will violate the selective decompression property as each encoded entry will not be of the same length. To preserve the selective decompression property, the jumps are encoded into a separate file. The jumps can be represented using tuples <*index, offset*> and then written to the file as 2 32-bit integers.

3.5 Reserved bits

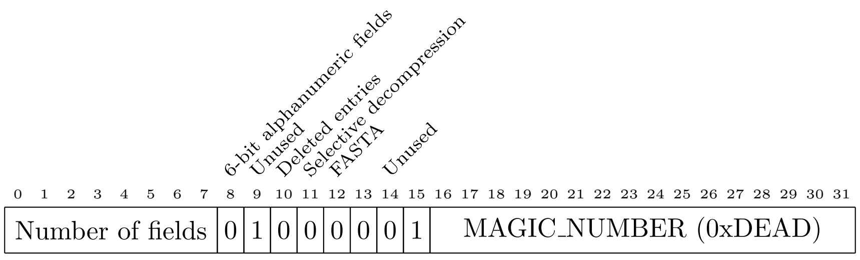
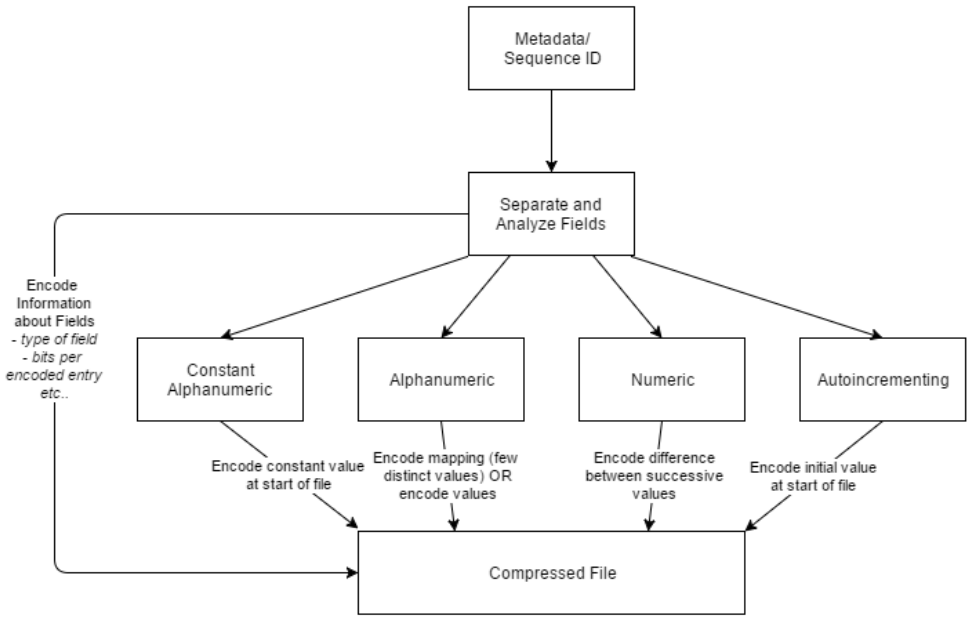


Figure 19: Reserved flags

Since there are still many unused bits in the first 4 bytes of the header, the last 16 bits will be provisionally allocated to fit a 2 byte magic number `0xDEAD`. This is useful to verify that the batch (see section 6.1 - Batch processing) of FASTQ metadata entries has been completely read. If after decompressing the metadata entries of a batch, after padding the read pointer to the next byte, then the 3rd and 4th byte should contain the magic number.

3.6 Summary

In this chapter, we have presented an compression algorithm for the metadata. Firstly, the metadata is tokenized into delimiter tokens and fields. We have determined four main types of fields, which are *Constant Alphanumeric, Alphanumeric, Numeric* and *Autoincrementing.* Then, based on the field type combined with the analysis of the metadata in the input file, an optimal encoding algorithm can be found.



Metadata/ Sequence ID
Separate and Analyze Fields
Encode Information about Fields
- type of field
- bits per encoded entry etc..
Constant Alphanumeric
Alphanumeric
Numeric
Autoincrementing
Encode constant value at start of file
Encode mapping (few distinct values) OR encode values
Encode difference between successive values
Encode initial value at start of file
Compressed File

Figure 20: Overview of Metadata Encoding Algorithm

Approaches that enable selective decompression (random access of compressed entries) is discussed thoroughly through the inclusion of various fixed-length codes. Moreover, methods to achieve further compression but do not allow selective decompression is presented.

Furthermore, to guarantee the integrity of the decompression process, in additional to the compressed metadata entries, the encoding of the compressed metadata stores information regarding the dimensions of the input files along with the optimal encoding methods chosen after the analysis of the input metadata.

4 Quality Scores Compression

This chapter tackles the problem of quality scores compression. Despite its chaotic and random characteristics, some patterns can be observed by the analysis of the quality scores. Various methods and models are explored. After careful evaluation, the optimal methods are chosen and incorporated into the compression tool.

4.1 Overview

As quality scores are seemingly random, it is impossible to obtain compression rates similar to those of the metadata and the sequences. Nonetheless, there are some characteristics of quality scores that can be exploited that can lead to a substantial compression.

Due to the differing quality score representations in the genomic data generated by different next-generation sequencing platforms, it is crucial to carefully develop a flexible algorithm that accepts different ranges of quality scores without fail.

4.2 Frequencies of quality scores

From the distribution of the symbols, it is immediately obvious that the frequencies of the symbols occurrence is not evenly distributed. Over half of the quality scores are one single value, i.e. 'H' in this file. The table containing frequencies is included in the appendix.

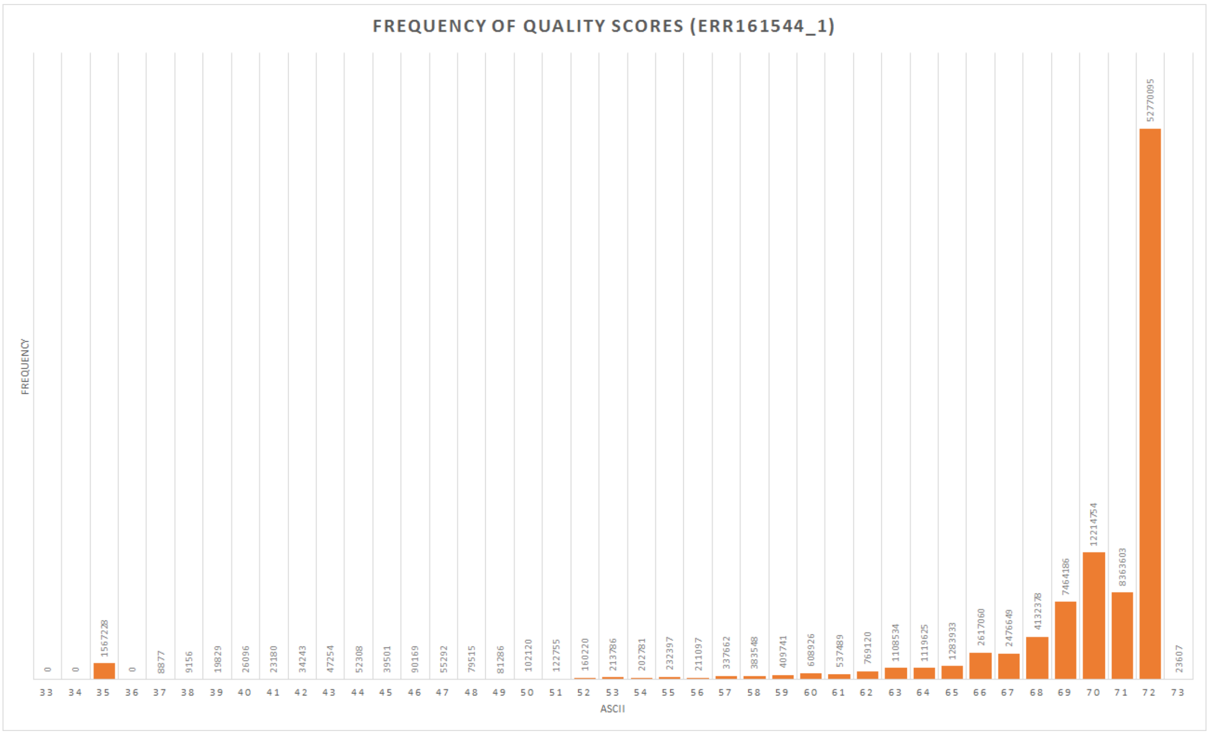


Figure 21: Frequency of quality score characters

Due to the uneven nature of the distribution, entropy encoding algorithms will work well to compress the data. From the frequencies of the first 100 million quality score characters, we can calculate that the entropy to be 2.6229. Thus, from just a good entropy encoding algorithm and compressing the characters straight up without further processing, we should be able to achieve the lower bound of 2.62 bits per quality score character. Using the fact that in its original ASCII encoding each character takes 8 bits, we can already achieve a compression ratio of 0.3.

Now we can attempt to find whether the occurrences of quality score characters are similar to what would be expected from a memoryless source. The memoryless property is when the previous and current state of a random variable does not affect the probability distribution of future events (in this case, the symbols). If the quality scores are memoryless, then entropy encodings such as Huffman compression are the only algorithms that will improve the compression ratio.

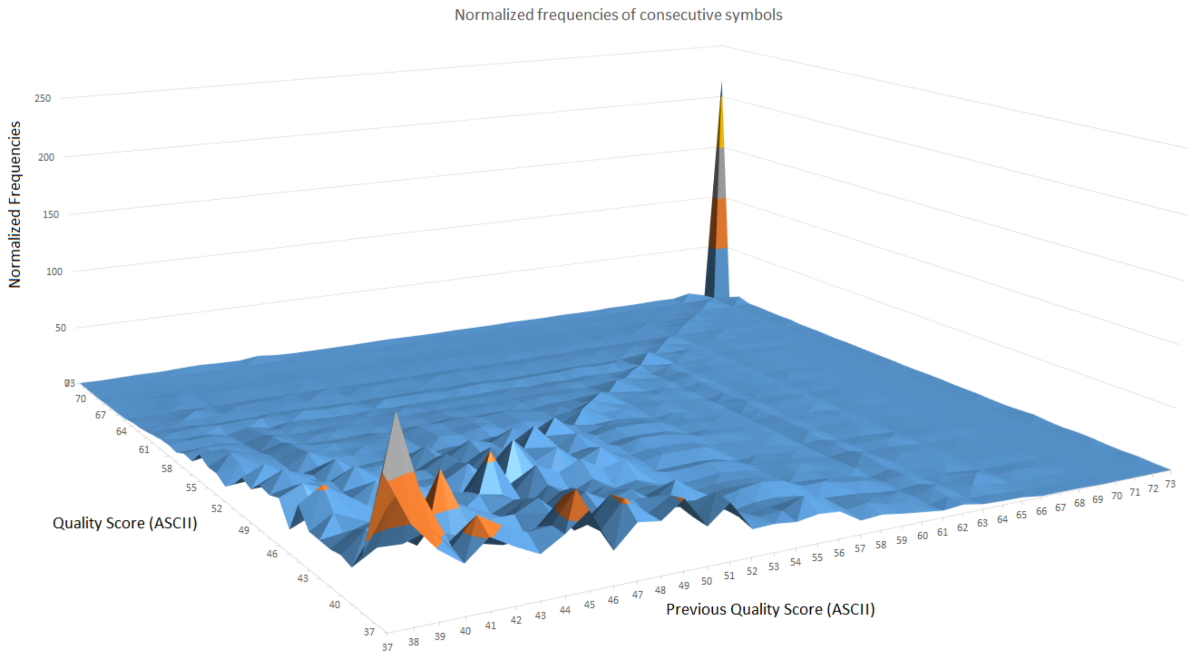


Figure 22: Normalized frequencies of consecutive quality scores

Figure 22 plots the normalized probabilities of the n-th quality score given the n-1 th quality score (i.e. $\frac{P(s_n|s_{n-1})}{P(s_n)}$). If the quality scores can be modeled as a memoryless random variable, then the contour graph should be almost flat. Nonetheless, it is obvious that there is a ridge down the middle where the n-1 th quality score is the same as the n-th quality score. This indicates that quality scores are not independent - and most of the time the next quality score is close in value to the previous one. Furthermore, we can find out that over 30% of the successive quality scores are repeated. Moreover, from the frequency table excerpt found in Appendix A, it can be observed that the probabilities are largely symmetric, i.e.: $P(s_n = x|s_{n-1} = y) = P(s_n = y|s_{n-1} = x)$.

By the above findings, it is obvious that we can further compress the result by processing the quality scores before utilizing entropy encoding.

4.3 Stochastic models

4.3.1 Linear Model

Next, we can find out whether we can accurately predict the next quality score based on the few previous quality scores by utilizing a stochastic model. First, we attempt to linearly interpolate a best fit from the five previous quality scores in order to the next quality score. Explicitly, we utilize linear regression and find the values/weights w_1, w_2, w_3, w_4 such that for the all quality scores q_n

$$(\sum_{i=1}^{4} w_i q_{n-i}) - q_n$$

is minimized. We choose to use the 4 previous quality scores to predict the next one as too many may cause overfitting while too few may not be sufficient to correctly model the quality scores. Using the first 100 million quality scores corresponding to the first million FASTQ entries, we find $w_i =$

$\{0.12335, 0.1779, 0.26597, 0.39339\}$. Now it is possible to entropy encode the difference between the predicted values and the actual values rather than just encoding the values which

Since the first million FASTQ entries were used as training data for the linear model, we use the next four million FASTQ entries to test how well it works.

4.3.2 Neural Network

As the linear model did not improve the entropy and thus in turn the compression rate, it is possible to look for a non-linear model. Now we choose a feedforward neural network with one input layer with 4 neurons corresponding to the four previous quality score symbols, one output layer with one neuron corresponding to the predicted quality score symbol. Now we determine the number of hidden layers and neurons by the rule of thumb as laid down in Jeff Heaton's *Introduction to Neural Networks in Java*, where one hidden layer is sufficient for nearly all problems, and the number of neurons to be the size of the input layer, which is 4 neurons.

Using **MATLAB**'s neural network toolbox, the neural network is trained using the first million quality score reads (from the sequence `ERR161544`), which is split into into three sets containing 50%-25%-25% of the data are the training, validation and test sets. The neural network is visualized in appendix A. For the hidden layer, the activation functions used are the *hyperbolic tangent*[29] function defined by

$$tanh(n) = \frac{e^{2n} - 1}{e^{2n} + 1} = \frac{2}{(1 + e^{-2n})} - 1$$

and for the output layer the activation function is just the linear sum.

Now that there is a trained neural network, the idea is to encode the first four values and then encode the differences between the predicted values from the neural network and the actual values in hopes that the entropy of the differences is less than that of the actual values.

Using the next million quality scores from the same sequence, which is once again one hundred million quality score characters, the differences between the 96 million (as the first four quality scores of each of the one million entries encoded) predictions and the actual values found, the entropy comes out to be 2.13, which is quite a huge improvement.

Although this is an improvement, there are a few things to consider. Firstly, the neural network is specifically tailored for this FASTQ dataset and may not be as good as predicting the values on other datasets. Secondly, the computation of each quality score requires 16 multiplications and 12 additions for the input weights, 4 additions for the biases in the hidden layer, four hyperbolic tangent calculations for the activation functions in the hidden layer, 4 multiplcations and 3 additions for the weights from the hidden to the output layer, and then finally one more addition for the output layer's bias.

The downside in the computational complexity and time required for this neural network prediction and encoding stage vastly outweights the 20% decrease in entropy and thus the output compressed size of the quality scores.

4.4 Modified Run-length encoding

The original run-length encoding transforms the plaintext into pairs of <*symbol, length*>. For a normal text file in ASCII, the worst case scenario is when there are no consecutive characters, and the file will be increased by the product of the number of characters with the number of bits to encode the length.

However, due to the fact that the domain of the quality score symbols do not span the entire ASCII space, we can utilize some of those characters to represent run-lengths instead. Thus we can first translate the original quality score symbols into the character set and symbols representing the run lengths *excluding the first character* (these run-length symbols will be denoted by $r_1, r_2, r_3 \cdots r_{len}$ where *len* is the length of the quality scores).

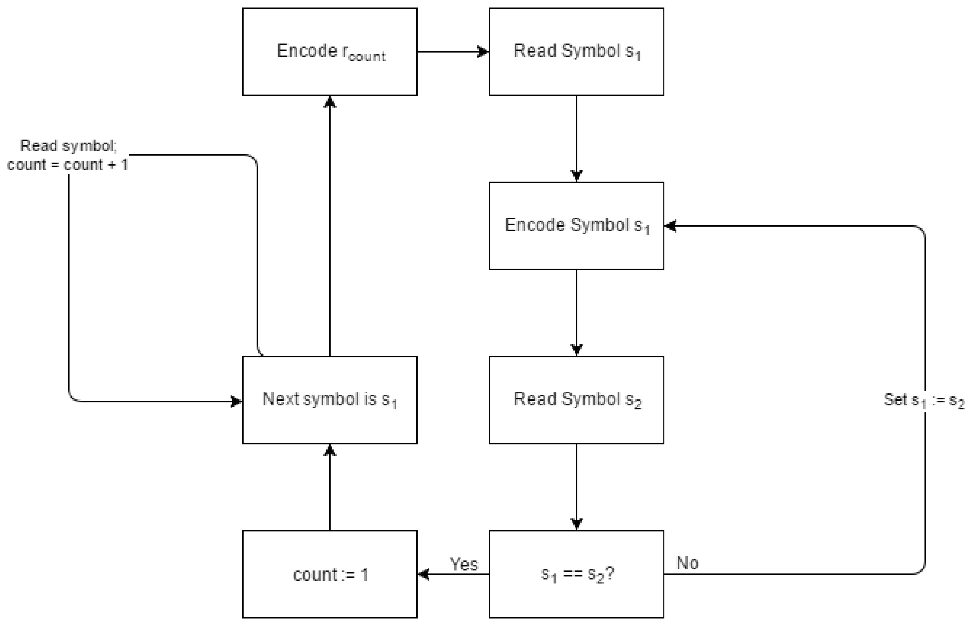


Figure 23: Diagram of the modified run-length encoding

4.4.1 Translating quality scores into run-length symbols

First, we read in a quality score symbol s_1. We find the corresponding ASCII value to the character, and subtract the ASCII value of the lowest quality score. For example, using the Illumina Casava v1.8 quality scores which contain 41 characters ranging from '!' to 'J', we subtract that ASCII Value of '!' (33) from all the quality score characters so that '!' maps to 0 and 'J' maps to 40. We then encode s_1. Next, if the next character s_2 is equal to s_1, we start a counter and set it to 1. We then read in $s_3, s_4, \cdots$, incrementing the counter by 1 each time a symbol is read until we reach s_n such that $s_n \neq s_1$ and $\forall i < n : s_i = s_1$. Now we can encode the sum of the maximum quality score's maximum value, which is 40 ('J') in this scenario, and the value of the counter.

Plaintext	A	A	A	B	C	D	D	D	D
Original Run-length	<A,3>			<B,1>	<C,1>	<D,4>			
Modified Run-length	A	r_2		B	C	D	r_3		

Table 11: Modified run-length encoding example

Now, the entropy of the symbols after applying the modified run-length encoding can be calculated using the same first 100 million quality symbol characters. The entropy is found to be 2.17, which is 2% worse than the neural network model. Nonetheless, as mentioned previously the neural network prediction requires significantly more arithmetic instructions per symbol and will require lengthly retraining on the input data to achieve that entropy. Thus, this modified run-length encoding is chosen instead.

4.4.2 Limiting the run-length

However, as mentioned previously, there is a cubic relationship between the number of symbols and the size of the encode/decode mapping between the 3-ary symbols to bits. Thus, while given the initial alphabet of 41 quality score characters, we have $41^3 = 68921$ mappings to consider, if we allow the run-lengths to reach the length of the quality score string entry, which can easily be greater than 100, the number of mappings (and thus the size of the map) will be $(41 + 100)^3 = 2803221$ which is 40 times greater than the original map.

Thus it is essential to limit the maximum run-length. As 99.9% of the runs are less than 20 characters, this is the limit chosen. The total number of mappings then becomes $(41 + 20)^3 = 226981$ which is a 13-fold decrease compared to allowing the run-lengths to be 100 or greater.

As we previously convert runs into the character followed by the symbol representing the run-length, as we have imposed a maximum, we will split the symbols corresponding to run-lengths of $k > 20$ into $\lfloor \frac{k}{20} \rfloor$ consecutive r_{20} symbols and then $r_{k \mod 20}$ if and only if $k \mod 20 \neq 0$. Although this increases the number of symbols where the run-lengths are extremely long, this still has negligible effects as the entropy per symbol decrease.

4.5 k-ary grouping

As we've established that there is a dependence between sequential quality score characters, grouping them should reduce the entropy per character, and thus increase the compression speed.

In theory, the larger the grouping, the lesser the entropy. However, when the groups are too large, the entropy will be artificially low - as in the extremes if the whole FASTQ file is treated as one group, then there will only be one symbol with 100% probability and thus an entropy of 0. This is not the only obstacle to the consideration of large blocks. If we have 41 quality score characters, then there are 41 possible symbols and encodings. However, if we encode using 5-ary groups, then there will be $41^5 = 115856201 \approx 1.16 \times 10^8$ symbols. To encode (and decode), we will have to store the information about the encoding of all these 100+ million symbols, which if each entry in the Huffman tree or the frequency counters in arithmetic encoding is just 32 bits, will require 3.7 gigabytes. While a 2-ary grouping's encoding information (Huffman tree or Arithmetic encoding frequency table) should take about 53 kB which is easily stored in the cache, for a 5-ary entropy code, even if 90% of the groups are cache hits (and thus 10% of entries being cache misses), the compression/decompression speed will suffer a huge performance hit from the memory accesses which will drastically overshadow the a theoretical maximum of 0.05 bits saved per quality score symbol compared to a 4-ary grouping entropy code or an approximately 13% maximum reduction in file size.

k-ary grouping	Entropy	Entropy / k
1	2.6229	2.6229
2	4.7126	2.3563
3	6.6138	2.2046
4	8.5421	2.1331

Table 12: Entropy of k-ary groupings of quality scores

Thus, balancing the benefits and consequences of the k-ary grouping entropy codes, we find that the 3-ary grouping seems to be the best fit for the quality score symbols out of the simplest entropy codes due to the fact that it should be largest grouping that the encoding table would fit into the cache of modern computer processing units. As the frequency distribution of characters become more and more uneven, the entropy decreases. This translation, however, distributes the high frequency of the symbols that occur the most to the newly created run-length symbols, thus increases the entropy. On the other hand, the *number of characters will decrease* as the previous runs of consecutive characters will become 2 characters instead. Thus to measure whether this is beneficial, we are not looking at the entropy nor the entropy per symbol, but the entropy $\times$ number of symbols = lower bound of number of bits necessary to encode. We find that the number of bits required to encode the data set using *only* 3-ary groupings is $2.2046 \times 100000000 = 2.2046 \times 10^8$. On the other hand, using our modified run-length encoding into 3-ary groupings will require $10.575 \times 19271844 = 2.0380 \times 10^8 < 2.2046 \times 10^8$, yielding a 8.5% improvement in compression ratio.

4.5.1 Grouping quality score symbols

So now our chosen algorithm to compress the quality scores is to first translate using our modified run-length encoding, then group the symbols 3 by 3 and then use an entropy code. This grouping is explicitly done by transforming the 3 consecutive symbols into 1 symbol via

$$S_n = s_{3n} \times \mathbb{A}^2 + s_{3n+1} \times \mathbb{A} + s_{3n+2}$$

where $\mathbb{A}$ is the number of sum of the run-length symbols (including the original quality score character set), S_n the grouped symbol, and s_n is the run-length symbols. This function is reversible as:

$$\forall n \in \mathbb{N} \Rightarrow \begin{cases} s_{3n} = \lfloor \frac{S_n}{\mathbb{A}^2} \rfloor \\ s_{3n+1} = \lfloor \frac{S_n}{\mathbb{A}} \rfloor \mod \mathbb{A} \\ s_{3n+2} = S_n \mod \mathbb{A} \end{cases}$$

4.5.2 Remaining symbols after grouping

As we have decided to use a 3-ary grouping for the symbols, the obvious problem is that there will be leftover symbols after the grouping when the quality score lengths are not exactly divisible by three. Now, as the length of the quality scores are known, the only option is to encode as if there are extra symbols with value 0. At the decoding stage, as we separate the 3-ary grouping into the individual symbols, we process the first symbol and check if after reversing the run-length encoding and appending to the results of previous symbols yields a quality score of length equal to that of the known length.

4.6 Arithmetic Encoding

Prior to the employment of arithmetic encoding, the entropy has been reduced as much as possible to increase the effectiveness of arithmetic encoding. Most implementations of arithmetic encoding utilize an adaptive approach, where each symbol's frequency is initialized to 1, and then while each symbol is encoded, the frequency of that symbol is incremented. Although this encoding method is the fastest, the compression ratio for the first symbols are very low, mainly due to the high number of symbols we have (226981 symbols) which causes the frequencies of even the most frequent symbols to be very insignificant until many symbols are read. When the frequencies are insignificant, then the subdivision of the <*low, high*> ranges mentioned in section 2.4.3 will be inefficient and the encoding of the frequent symbols will still lead to a much higher number of bits than expected from the overall distribution and entropy. Thus, it is more optimal to find all the frequencies for the symbols in the file before encoding. To make decoding possible, this table of frequencies will need to be stored in the compressed file as well. This has two main benefits - one is that the compression ratio will improve significantly (which is especially apparent in relatively small files). The other benefit is that the decompression speed will be much quicker as we do not need to increment the frequencies for each decoded symbol since the total frequencies are known prior to the decoding. However, the obvious downside is that the compression speed would be impacted.

Furthermore, instead of compressing the symbols from each FASTQ quality score entry separately, we can use the previous <*high, low*> values in the previous entry which allows us to not delineate between FASTQ quality score encoded entries by encoding an EOF (end-of-file) symbol. As the frequencies for the EOF symbol is very low, it will require a significant amount of bits to encode. Taking into account the number of reads can easily reach tens or hundreds of millions, it is obvious that removing the need for the EOF symbols will decrease the compressed file size.

4.7 Encoding of Quality Scores

In addition the the final output of the compression pipeline for the quality scores described above, to guarantee the integrity of the compressed files and to speed up the decompression process, we encode some additional information. Firstly, we encode the number of FASTQ entries (and thus the number of quality score entries). Secondly, we also encode the length of each FASTQ entry, which we assume is consistent throughout all the reads. These two pieces of information will allow the decompression process to be sped up as the decompression process will be able to distinguish which FASTQ entry the decoded symbols belong to. Furthermore, we need to encode the character corresponding to the lowest quality score. This is due to the fact that different sequencing platforms use different ranges of characters to denote the quality scores. Moreover, the number of quality score characters needs to be encoded. The number of quality score characters is value of the maximum quality score character found subtracted by the value of the minimum quality score character. Since the quality score characters are all encoded in ASCII or UTF-8 in the original FASTQ file, 8 bits will be sufficient to encode both the base quality score character and the number of quality score characters. However, to allow for future compatibility and possible extensions into UTF-16, 16 bits are allocated for both values. Finally, we include a 32-bit magic number at the end to mark the end of the file. Thus, our overall file structure is as follows:

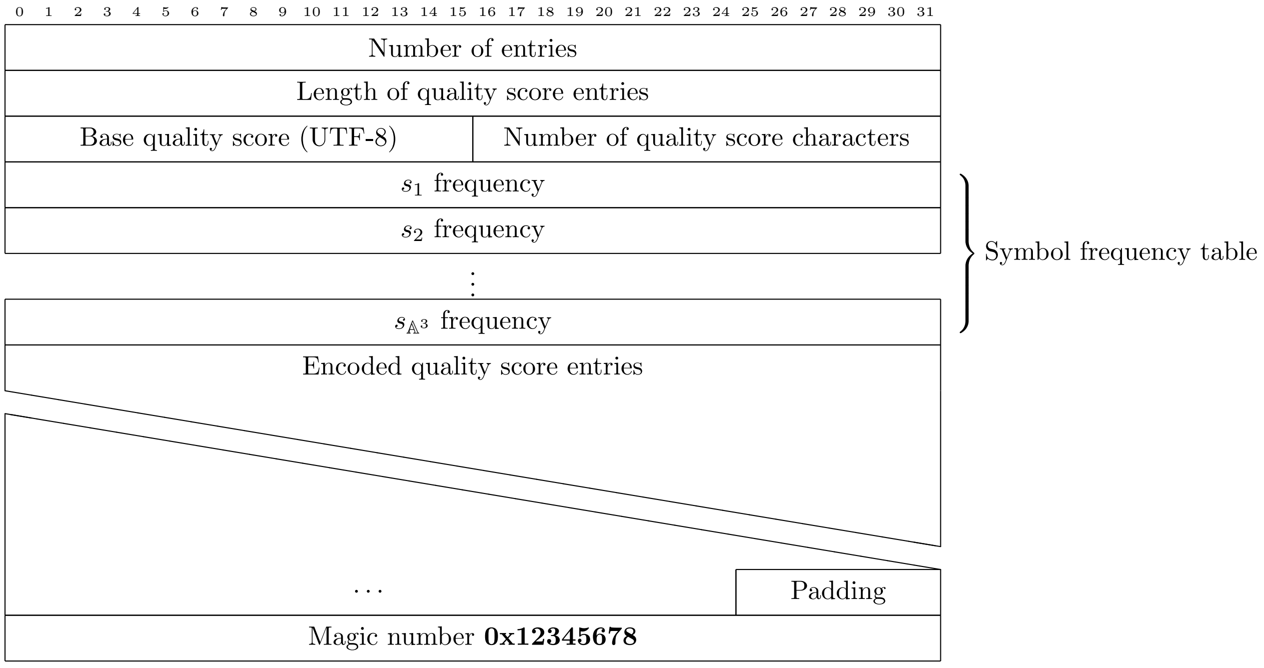

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Number of entries
Length of quality score entries
Base quality score (UTF-8)
Number of quality score characters
s_1 frequency
s_2 frequency
$s_{\mathbb{A}^3}$ frequency
Symbol frequency table
Encoded quality score entries
...
Padding
Magic number **0x12345678**

Figure 24: Encoding structure of Quality Scores

4.8 Summary

In this chapter, we present a compression algorithm for quality score compression after analyzing patterns in the quality scores. The algorithm is composed of three stages: firstly, we utilize a novel version of run-length encoding which is effective due to the observation of frequent equal consecutive characters. Secondly, we group the symbols from the quality score encoding 3-by-3 as we find that subsequent quality scores are likely to be close to each other in value, thus reducing the entropy of the symbols. Finally, we employ arithmetic encoding, which is a form of entropy encoding that reaches a compression ratio similar to the theoretical maximum given the frequency distribution of a set of symbols. Entropy encoders are effective as the probability distribution of the quality score characters, thus the ternary (3-ary) groupings, are disproportionate.

5 Final Stages of Compression

As the compression algorithms for three of the four FASTQ components have been determined above, this chapter will present the encoding of the last remaining FASTQ component, the *strand*. Furthermore, the four encoded files will have to go through a final stage of compression and then combined into a single file. This chapter concludes the development of the compression and decompression methods for our compression tool.

5.1 Strand

The last component yet to be encoded is the strand, which is arguably the simplest component. There are four possible formats for the strand. Firstly, the first character can either be '+' or '-'. Secondly, the metadata line excluding the initial '@' character may *optionally* be copied after the '+' or '-' symbol. Thus, the strand simply need 2 bits to encode the information necessary for these four cases.

Moreover, as the format for strands is consistent throughout the file, this 2 bits will only be required to be encoded once instead of once per entry. As memory is not bit addressable (in most operating systems), it is required to encode these 2 bits with a full byte.

This stage is simply done by reading the 3rd line of the FASTQ file, which is the strand of the first FASTQ entry. The procedure involves initializing a byte sized type to 0. If the length of the line is equal to one, we set the 2nd least significant bit. Then, if the first character is a '+', we set the least significant bit.

However, there is a slight limitation where most if not all file systems have block sizes much larger than a byte. This causes a problem as file sizes must be an exact integer multiple of the file systems block size. In the default file system of Linux, which is `ext3` or `ext4`, the block size is 4 kilobytes, or 4096 bytes. Still, this will still be insignificant in terms of the compressed file size even for small sequence datasets.

In the decompression stage, the one byte is first read from the strand file. The two least significant bits are checked, and then the '+' or '-' symbol, possibly followed by the repeated metadata, is simply interleaved with the rest of the decompressed FASTQ components to form exactly the original file.

5.2 Intermediate files

So far, each of the FASTQ components are compressed into different intermediate files. Given the original filename *DATASET.fastq*, the following intermediate files are created:

- Metadata
  - *DATASET.fastq.md* - the compressed metadata
- Quality Scores
  - *DATASET.fastq.qs* - the compressed quality scores
- Sequence
  - *DATASET.fastq.cnt* - the number of tuples of each sequence
  - *DATASET.fastq.pos* - the **position** values of the tuples
  - *DATASET.fastq.len* - the **length** values of the tuples
  - *DATASET.fastq.sym* - the **symbol** values of the tuples
  - *DATASET.fastq.orn* - whether the reads are reverse complemented
- Strand
  - *DATASET.fastq.str* - the 1-byte file encoding the strand

Although it may seem surprising how many intermediate files there are especially for the sequence compression part, this is purposely done to enable the next stages of the compression.

5.3 pbzip2

After the sequence and quality scores are compressed, using the methods described above, an additional layer of compression is applied. This last layer of compression may not yield incredible results, but the tradeoff is minimal if the compression does not require a long time. Thus, **pbzip2**, a parallel implementation of the popular **bzip2** compression algorithm was chosen. This stage compresses the sequence and quality scores using the default settings of **pbzip2**. The strand and the metadata are not compressed at this stage due to the fact that the strand's filesize is a single byte and there is no compression tool that would yield a smaller filesize, and that the metadata is very densely packed which empirical results show that the reduction in filesize is very minimal. This stage is also the reason why the sequence compression creates five intermediate files. As **pbzip2** compresses better as the data is similar, it would work best applied independently to all the separate encoded values.

5.4 tar

The final stage of the compression is combining all the different compressed components into one single archive. This allows for more organized access and distribution of the data. **tar** is run with the flags *c* (create) and *f* (file), and then the output file is briefly named *temp*. This allows the compressed components such as *DATASET.fastq.qs.bz2* and *DATASET.fastq.md* which are intermediate products of the compression to be all removed with the POSIX command *rm DATA.fastq.**. Finally, the *temp* temporary file is renamed to *DATASET.fastq.tar.bz2*.

5.5 Combining the components

The FASTQ file is first read and each entry separated into its components and then represented using the following structure.

```
struct read_t {
  char meta_data[MAX_META + 1];     // MAX_META = 100
  char sym[MAX_SEQ + 1];            // MAX_SEQ = 150
  char strand;                      // '+' or '-'
  char q_score[MAX_SEQ + 1];
  uint8_t seq_len;                  // Length of sequence
  uint8_t meta_len;                 // Length of metadata
  bool unknown;                     // Whether the sequence contains unknown smybols
};
```

An array of this structure is then passed to all the individual compression algorithms. Since after this file input, the read structures will not be modified by the functions, the array can be passed to the compression functions for the different components to be processed at the same time.

5.6 Summary

Now to summarize, in this chapter we have described the encoding structure of the final FASTQ component - the *strand*. Then, all the compressed components of the FASTQ file are put together into one archive following the final *bzip2* stage of compression for the quality scores and sequences. The graphic visualization of the complete pipeline of the compression algorithm from figure 1 is copied onto the next page for the reader's convenience:

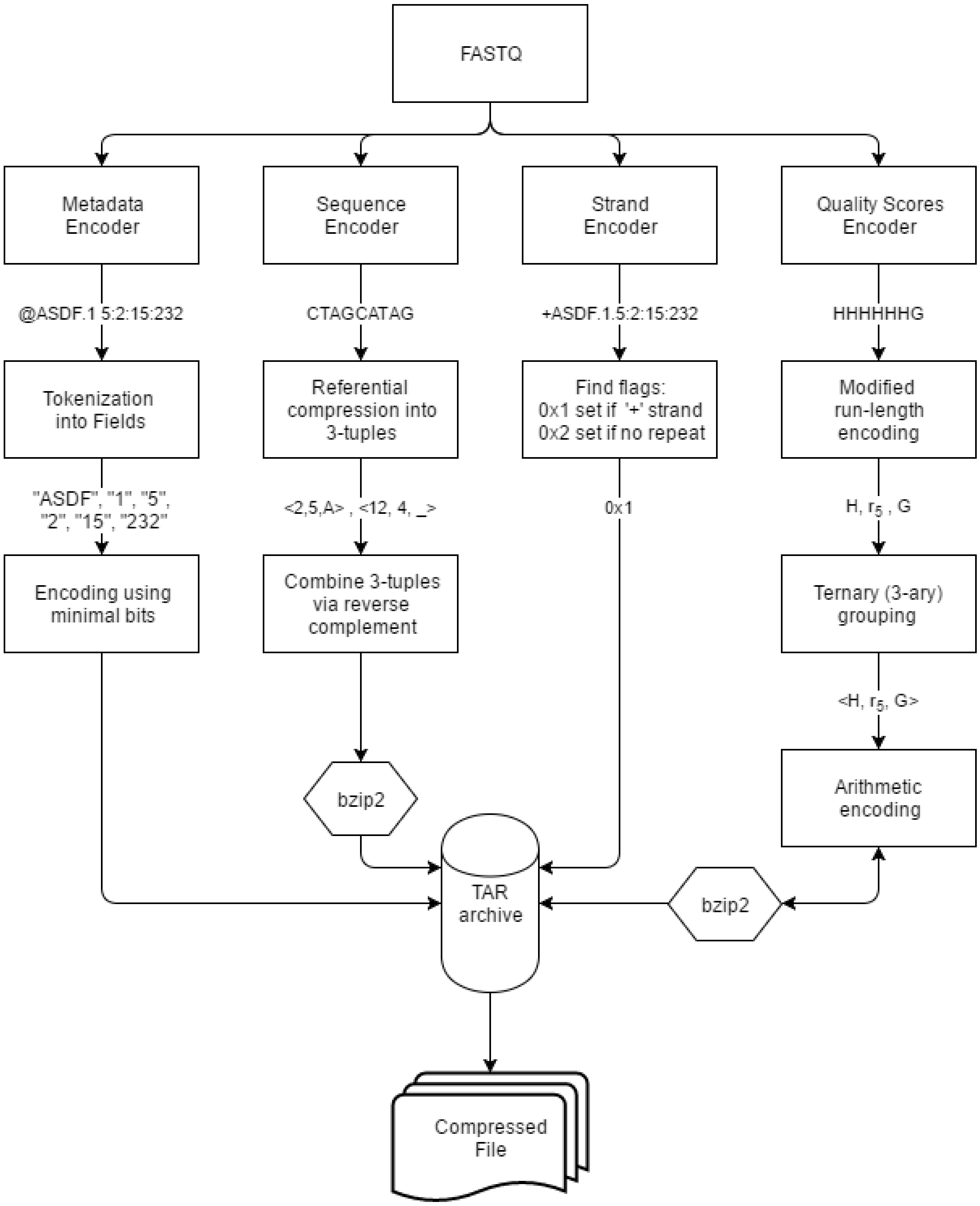


Figure 25: Overall pipeline of compression algorithm

6 Acceleration

In the previous sections, the algorithms utilized to compress the different components of a FASTQ (and FASTQ) dataset is determined. Even though the algorithm achieves the first aim of provided a compression utility of high compression rate, it is crucial to have low compression and decompression times.

Most of the FASTQ compression tools in existence can take up to tens of minutes or even hours to compress a larger FASTQ file such as the human genome with three billion base pairs (the benchmarking against these tools is done in the next section). As these FASTQ files are huge in size and can easily reach tens of gigabytes, parallelization of the algorithm is crucial to reducing the compression time to acceptable levels.

This section provides the following contributions:

1. Parallel execution of file input and output with the compression algorithm to maximize the time spent compressing the genomic data using threading and the subdivision of input data into contiguous batches;
2. Acceleration of the metadata compression by the OpenMP parallelization of the compression of metadata in subdivided input segments and a greedy approach in the analysis of metadata tokens/fields when determining the encoding process by reuse of results computed from previous batches; and
3. Acceleration of quality score compression by the OpenMP parallelization and the reusing of arithmetic encoding cumulative frequency tables from previous batches.

6.1 Batch processing

The naive way to run the algorithm would be to read the whole file and perform the compression process on it. However, using a batch processing system, one thread can handle the file I/O can be read to/written from a buffer while another compresses **and** decompresses the contents. Furthermore, the whole file would not be required to be read to RAM before processing, which prevents the overhead time for page swaps when the file does not fit into RAM. Furthermore, as the individual compression methods for each of the FASTQ components are independent, a thread can be created for each.

The size of each batch is user-defined, and defaults to 400MB. The first problem arises as when the contiguous memory is copied into the buffer during compression, the last FASTQ entry may not be completely read into the file. Therefore, the allocation of the buffer reserves an extra 100KB of space to store the remaining section of the FASTQ entry. Then, the FASTQ file is read byte-by-byte until a newline '\n' followed by an '@' symbol is read. As mentioned previously, the first line of the FASTQ file, the metadata/sequence identifier, begins with an '@' symbol. Thus, copying up until the newline symbol will ensure that the whole FASTQ entry is being read.

A few modifications to the algorithm are necessary. To simplify the process, each batch is encoded as if it was an independent FASTQ file. This requires some additional overhead as the headers for the metadata and the quality scores will be repeated.

6.1.1 Quality Scores

While the metadata headers are insignificant in terms of size relative to the compressed metadata entries, the quality score's serialized frequency table containing approximately 230000 32-bit integers takes over 900KB of space per batch, which is significant as using the default batch size of 400MB yields approximately 160MB of quality scores, and using our previously calculated entropy values of 2.1 (i.e. 2.1 bits per 8 bit ASCII character) implies that the compressed quality score values for the batch is 40MB. Furthermore, calculating the cumulative frequency table repetitively accounts for more than 70% of the compression time.

6.1.1.1 Recurrently Recomputing Frequencies

The first option is to allow the user to set a value `FREQ_BATCH_REFRESH` representing the number of batches per frequency table. The frequency table will be calculated using the first batch, and will be utilized for the subsequent `FREQ_BATCH_REFRESH - 1` batches. The frequency table will thus only be written to the file every `FREQ_BATCH_REFRESH` batches. It is should be noted that due to the number of possible 3-ary grouped symbols, if `FREQ_BATCH_REFRESH` is high, then subsequent batches will have a worse compression ratio than if no grouping was utilized. Thus, in a batch encoding system with frequency table sharing, 3-ary grouped symbols may not be actually optimal especially when the batch's grouped symbols are not 2 orders of magnitude greater than the total number of symbols. Thus, with this in mind, the solution must be to not employ the grouping during the compression algorithm if `FREQ_BATCH_REFRESH` is greater than 1 or the batch size is less than 250 times the number of 3-ary symbols. Now, we can put the `MAGIC_NUMBER` at the beginning of each encoded batch of quality scores, thus detecting if the arithmetic compression's integrity has failed as it will overwrite the `MAGIC_NUMBER`. Furthermore, we need to set a bit to denote whether this batch is accompanied with the frequency tables or the decompression algorithm will treat the first batch of arithmetic encoding values as the frequency tables, possibly causing segmentation faults (when the entries are not completely decoded but the end of the file is reached) or arithmetic exceptions (when the frequency is read as zero). The value will be encoded in the 65th bit of each compressed batch (if the bit is set then the frequency table is included). This restricts the length of the quality score entries, which is the length of a single FASTQ read, to $2^{31} = 2147483648$. However, current next generation sequencing machines usually have a read length of 50 to 120, and thus 31 bits should be more than sufficient for the foreseeable future.

6.1.1.2 Greedily Recomputing Frequencies

The other option is to let user set a threshold for the recomputation of the frequency table. The threshold, `THRESHOLD_BITS_PER_10Q` is an unsigned integer representing a number of bits per 10 quality score characters. Thus, using this greedy approach, each batch (except the first batch) is encoded using the previous symbol frequencies. If and only if the threshold is crossed, i.e.

$$\frac{10 \cdot \mathit{file\_size}}{\mathit{entries} \cdot \mathit{entry\_length}} > \mathit{THRESHOLD\_BITS\_PER\_10Q}$$

then the frequencies are recomputed and the symbols recompressed.

Note that the information about the quality score character ranges needs to be reencoded as a quality score character may not exist in the first batch but could possibly exist in the subsequent batches. The modified encoding structure of the compressed quality scores is shown in figure 26.

6.1.2 Metadata

To increase the compression speed, a greedy approach is utilized in the batch encoding of the metadata. After the first batch analyzes the fields in the metadata and sets the bit-width for each field, subsequent batches utilize the same encoding format as the previous batch. Obviously, some batches may have a value that exceeds the maximum value allowed using the encoding format of the previous batch. Only then the metadata's values are reevaluated to determine the new encoding format. This greedy method works best when the batch sizes are significantly large as sequential batches should mostly have the same ranges of values. As finding the ranges for the metadata fields is responsible for most of the time used to compress the metadata, even at the worst case where each batch requires a new set of encoding formats, the compression time for the metadata should not increase by more than 10%. Moreover, as the metadata compression (including the analysis of the ranges) requires less time than the compression for both the sequence and the quality scores, this greedy algorithm will not become the bottleneck for the overall compression scheme.

Furthermore, the analysis of the metadata fields can be parallelized. The method chosen is similar to the MapReduce model. Firstly, we separate the metadata entries into `N_THREAD` sections. Now, the previously outlined procedure to analyze the metadata and outputs a `MetadataFieldType` is modified to return a `struct MetadataAnalysis` containing a `MetadataFieldType type`, an unsigned 32-bit value

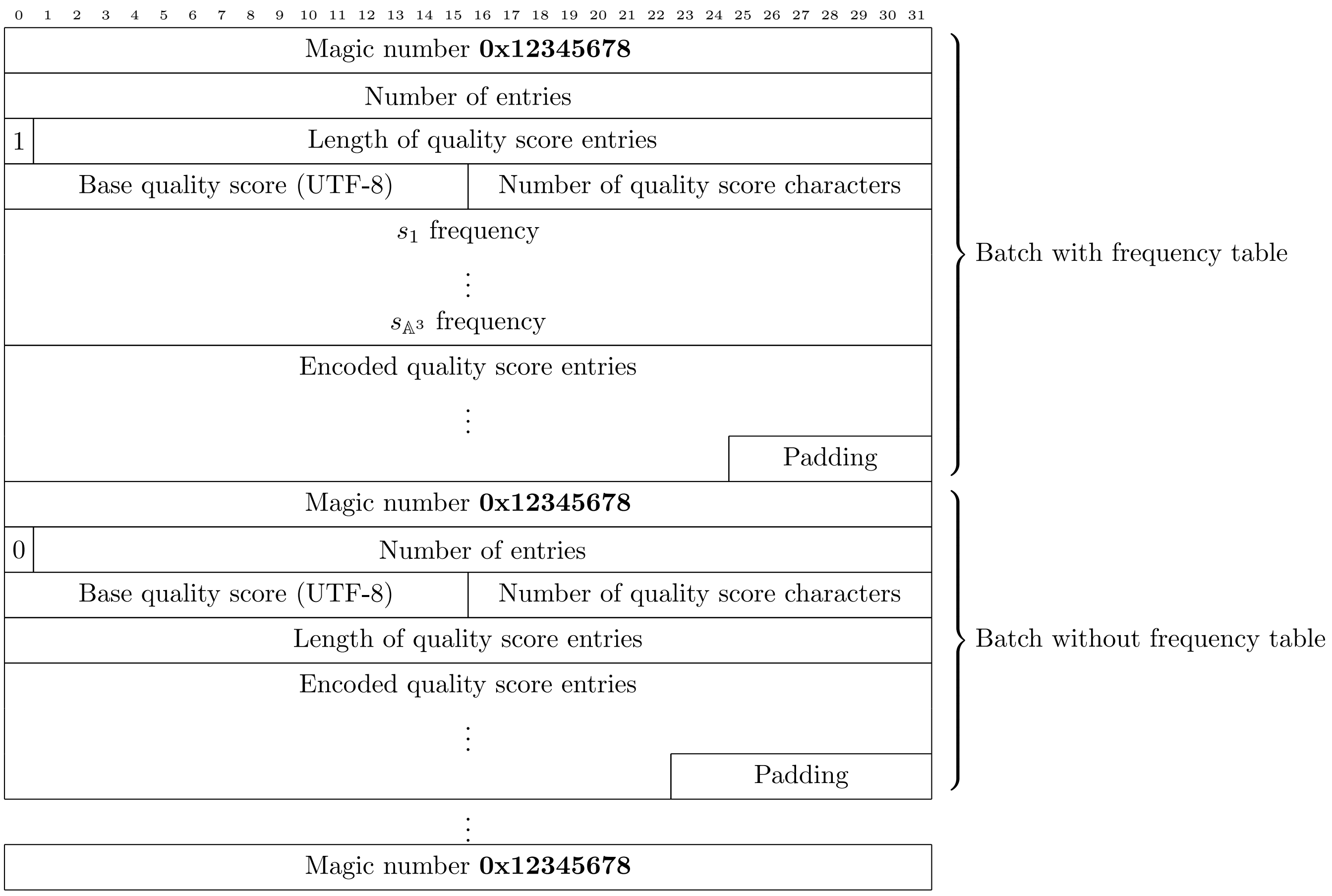

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Magic number **0x12345678**
Number of entries
1
Length of quality score entries
Base quality score (UTF-8)
Number of quality score characters
s_1 frequency
⋮
$s_{\mathbb{A}^3}$ frequency
Encoded quality score entries
⋮
Padding
Batch with frequency table
Magic number **0x12345678**
0
Number of entries
Base quality score (UTF-8)
Number of quality score characters
Length of quality score entries
Encoded quality score entries
⋮
Padding
Batch without frequency table
⋮
Magic number **0x12345678**

Figure 26: Encoding structure of Quality Scores for Batch Encoding

MetadataAnalysis A.type	**MetadataAnalysis B.type**	**Reduced MetadataAnalysis value**

Constant alphanumeric	Constant alphanumeric	if a.value = b.value
		<Const. alphanumeric, a.max, a.value >
		else if isnumeric(a.value) and isnumeric(b.value)
		<Numeric, MAX(int(a.max), int(b.max)), _ >
		else
		<Alphanumeric, MAX(a.max, b.max), _ >

Alphanumeric	Alphanumeric	<Alphanumeric, MAX(a.max, b.max), _ >
Alphanumeric	Constant alphanumeric	
Autoincrementing	Autoincrementing	<Autoincrementing, MAX(a.max, b.max), _>
Autoincrementing	Alphanumeric	<Alphanumeric, MAX(log_{10}a.max, b.max), _ >
Autoincrementing	Constant alphanumeric	
Numeric	Alphanumeric	
Numeric	Constant alphanumeric	
Numeric	Autoincrementing	<Numeric, MAX(a.max, b.max), _ >
Numeric	Numeric	

Table 13: Reduction rules for **MetadataAnalysis**

`uint32_t max` which represents the maximum numeric value in a numeric (or auto-incrementing) field or the maximum length of the longest alphanumeric string for an alphanumeric (or constant-alphanumeric) field, and `std::string str` which is the unique value if the field type is constant alphanumeric, and `NUL` otherwise. Then, the rules for the reduction part are formulated in table 13. Note that the reduction rules are symmetric.

6.2 OpenMP

As OpenMP is widely supported on most modern CPUs and operating systems, the usage of OpenMP in this compression utility will not require the user to further install or configure anything. In the multitude of compression methods utilized, many can be modified such that the algorithm can be partially parallelized. However, it is impossible to parallelize the file I/O portions due to the fact that the order of the encoded entries must be preserved from the order of the original FASTQ file. Thus, modifications must be made such that the FASTQ entries in the aforementioned `read_t` structures can be encoded to an intermediate state. This encoding stage should be able to be parallelized, which then the intermediate states are written to the file.

Now, first we have to determine the number of threads and specified. We allow the user to define a variable `N_THREADS` which is then passed to the OpenMP during the preprocessor stage

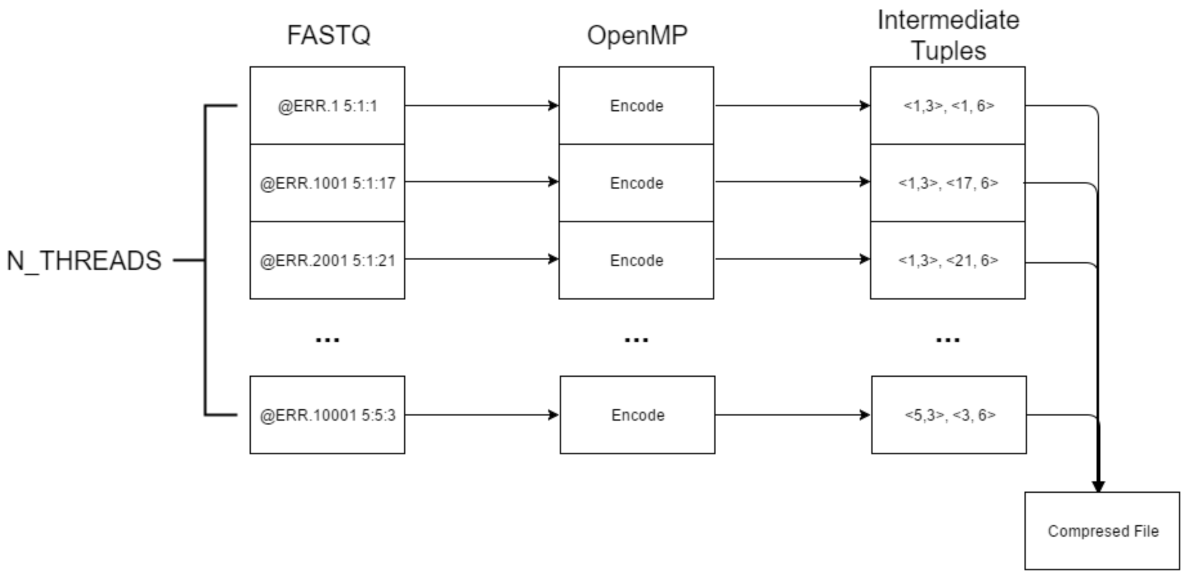


Figure 27: Parallelized Metadata Compression

6.2.1 Metadata

The main metadata encoding loop can be parallelized easily. However, as the file writes need to be sequential, we first separate the metadata entries into `N_THREADS` contiguous blocks of entries. Then, an intermediate state is defined as a tuple <*value, width*> which corresponds to the value and width of each encoded field of each entry. Using these tuples, the encoded entries are then sequentially written to the compressed file.

If no non-fixed length encodings are used, i.e. when selective decompression is enabled, this property can allow the decompression stage to undergo a similar parallelization. As each encoded FASTQ metadata entry is of fixed-length which is known due to the width of each field being encoded at the start of each encoded batch of metadata, the total width $w_{md} = \sum_{f \in fields} w_f$ of each encoded FASTQ metadata entry can be found. Now each of `N_THREADS` OpenMP threads $t_i \in t_0 \cdots t_{N\_THREADS}$ will seek to $w_{md} \times i / batch\_size$ and decode $batch\_size/\text{N\_THREADS}$ entries.

6.2.2 Quality Scores

The first two stages of the quality scores compression, which is the modified run-length encoding and optionally the 3-ary grouping if enabled by the user, can be done through OpenMP. As the results of these two stages for each entry are independent of each other, an array of size equal to the number of reads and each entry is an array of integer values corresponding to the run length symbols or their 3-ary grouping counterparts. The FASTQ reads are the run-length encoded and grouped and placed into the corresponding array index.

During this stage, the frequencies of the symbols are also summed in this stage, which then is calculated into the cumulative frequency table for the arithmetic encoding. As the translation from the quality score strings into the 3-ary grouped symbols accounts for half the time spent on the compression of quality scores in the previously linear and sequential process, there is a vast reduction in the time utilized. Nonetheless, the quality scores compression for the human genome (`ERR161544_1.fastq` - 19GB file) still takes 400 seconds with 16 OpenMP threads, albeit down from a previous 750 seconds.

Now, an attempt to parallelize the arithmetic encoding process is described. Although arithmetic encoding cannot be truly parallelized due to its aforementioned encoding process consists of encoding the arbitrary precision floating point value corresponding to the input string, it is possible to split the quality scores into subbatches and encoding them in parallel. Obviously, the file I/O must be sequential, but using 16 OpenMP threads, the time it takes is reduced from 400 seconds to 70 seconds. However, we now need to modify the encoding structure once again to ensure the information required to reverse the encoding process is provided and moreover for integrity purposes. Firstly, the number of subbatches `N_THREADS` will require to be encoded as this user-defined parameter may not be consistent across different copies of the compression tool. Secondly, the size of each subbatch is encoded to make the decoding process. During the decoding (and encoding) process, the subbatch sizes can be summed and compared to the encoded number of total entries in that batch to ensure no corruption or missed entries. Finally, the `MAGIC_NUMBER` is encoded in between every subbatch to clearly delineate between the subbatches and checking whether the previous subbatch was properly encoded in the encoding process.

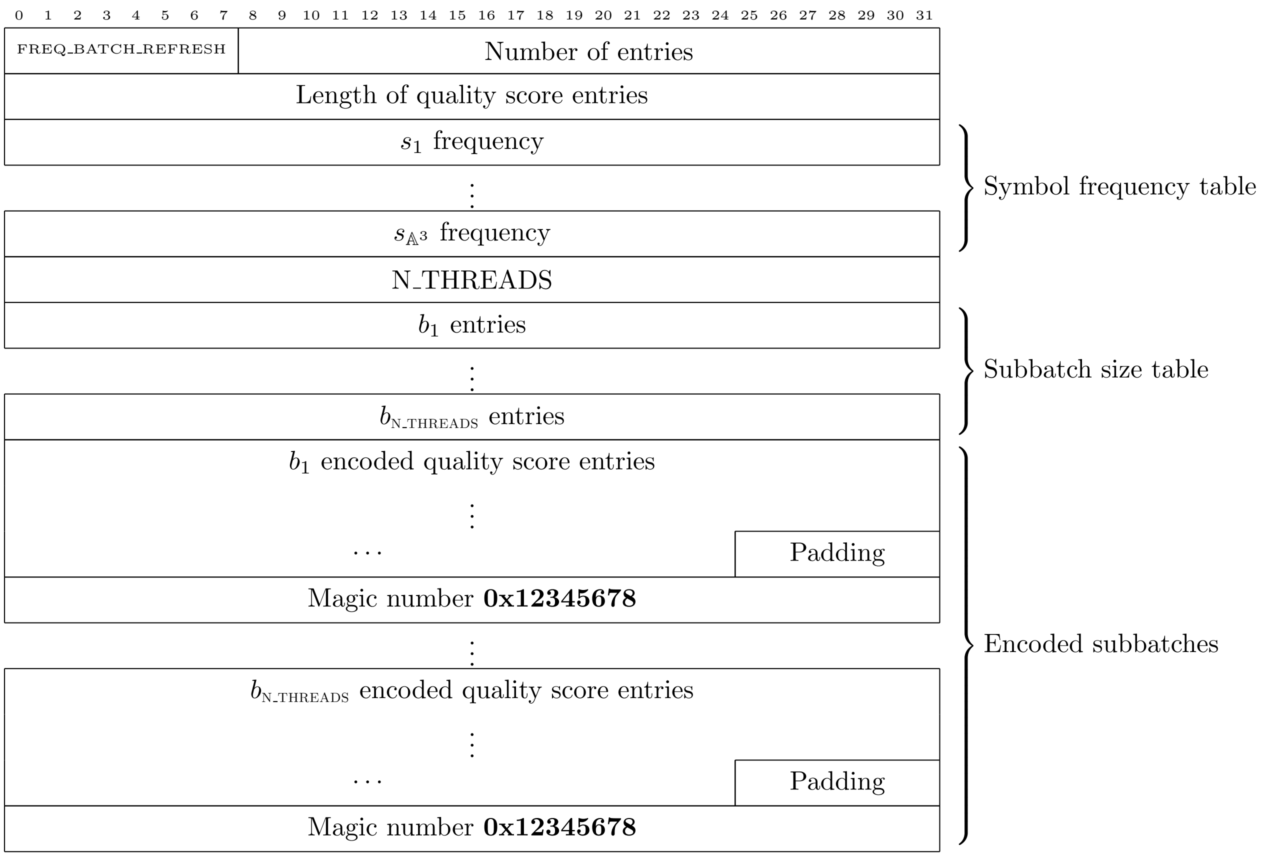

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
FREQ_BATCH_REFRESH
Number of entries
Length of quality score entries
s_1 frequency
$s_{\mathbb{A}^3}$ frequency
Symbol frequency table
N_THREADS
b_1 entries
$b_{\text{N\_THREADS}}$ entries
Subbatch size table
b_1 encoded quality score entries
Padding
Magic number **0x12345678**
$b_{\text{N\_THREADS}}$ encoded quality score entries
Padding
Magic number **0x12345678**
Encoded subbatches

Figure 28: Encoding structure of Quality Scores with parallelization

6.2.3 Sequence

The compression of the sequences can be parallelized similar to the metadata as both of the compression algorithms are independent of previously encoded information. An array of vectors of tuples is created with the array size equal to that of the number of reads in the batch. Then, the algorithm just finds the vector of tuples corresponding to the mappings of the reference to the sequence and the sequence's reverse complement, and then the vector with the lesser number of tuples is stored in the array. Then, this array of vector of tuples is written to the output file.

6.3 Summary

In this chapter, we have presented various novel modifications to the compress algorithm described in the previous sections:

- The input file is split up into batches, which can reduce the maximum memory usage in addition to allowing the parallel execution of the compression of one batch while performing file input/output functions on the previous batch. This led to slight modifications of the encoding structure of the compressed metadata and quality scores to allow seamless batch decompression;
- Novel greedy algorithms in the determination of the optimal encoding structure of the current batch. As the analysis stage of the quality scores and metadata to figure out the frequencies of various symbols and the ranges of values of different fields requires more time than the actual encoding itself, a greedy algorithm to reuse the encoding structure of the previous batch is implemented. The metadata is only reanalyzed if a FASTQ entry not losslessly compressable using the previous encoding structure is encountered. Moreover, the quality score frequencies of previous batchs are reused until the compressed size of the resultant batch is above a user-definable limit;
- Further subdivision of batches to parallelize the compression of quality scores and metadata; and
- The implementation of the sequence compession on the Maxeler dataflow computing engine, which dramatically reduces the sequence compression time up to 4-fold.

7 Benchmarking

To demonstrate how the compression tool **compress** outlined above compares to other existing compression tools, in this chapter, **compress** is benchmarked against six other compression tools using six data sets. The benchmarks measure: (1) the compression ratio, (2) the compression speed, and (3) the decompression speed. All these datasets are in the FASTQ format, as many of the genomics domain specific compression tools are tailored towards FASTQ. Finally, the compression tool's memory usage is evaluated.

7.1 Selected datasets

As mentioned previously, our tool is targeted towards the high quality and coverage sequencing data, and more specifically, the human sequences that are the challenge and one of the main points of interest in the field of genetics. Thus, the first dataset we choose is the sequence `ERR165144`. Then, five additional datasets used in the evaluation section of the paper published with the **LWFQZIP** compression tool are chosen. All six of the datasets contain sequences of a different species. Moreover, there are a wide range of read lengths and read counts (and ergo file sizes), which should demonstrate how the different compression tools compare given different inputs.

Finally, the reference sequence genomes for reference-based compression are to be chosen. In this case, for all the six datasets, another dataset containing the same species as the sequence to be compressed is chosen as the reference. These reference sequences are then processed and the FM-index generated. Although technically the complete human genome would work as a reference for all the datasets, loading the reference dataset may skew the results for compression speed and decompression speed as the overhead for loading the reference index may be greater than the time utilized to compress a FASTQ file of small size. If every single compression tool benchmarked employed reference-based compression, then the relative results should be unchanged. Nonetheless, three of the compression tools used in the following benchmark does not use reference-based compression, thus it is necessary to use different reference genomes.

The datasets chosen, in no particular order, are listed in the table below:

Dataset	Species	Read Count	Read Length	File Size / GB	Reference
ERR231645[9]	E. coli	6344039	51	1.1	NC_000913
SRR801793[28]	L. pneumophila	5406461	100 * 2	2.8	NC_007005
ERR233152[10]	P. aeruginosa	2745192	77	0.58	AP014622
ERR005143[7]	P. syringae	3551133	36 * 2	0.94	NC_007005
SRR327342[27]	S. cerevisiae	15036699	138	4.9	ACFL01000033
ERR161544[8]	H. sapien	74111640	100 * 2	38	hg38

Table 14: FASTQ Datasets used for benchmarking.

Note that if '* 2' appears in the read length, the dataset is a paired-end sequence. A paired-end sequence[19] is the result of DNA fragments being sequenced from both directions, which generate high quality and alignable datasets. For the benchmarking, both FASTQ files will be compressed.

7.2 Benchmarking process

The compression and decompression are all run on a machine with the following specification:

- Processor: *Intel(R) Xeon(R) CPU E5-2640 v0 @ 2.50GHz (6 cores)*
- Random Access Memory: 64GB
- Operating system: CentOS v6.4.
- Main memory: 500 GB solid state drive

All the compression tools are run with their default settings with the exception of **quip**. The assembly method is not enabled in the following benchmarks of **quip** and its referenced-base compression is used instead. This setting is due to the fact that there is little to no improvement in the compression ratio shown despite a significant increase in compression time for the datasets above.

7.3 Compression ratio

Using the aforementioned datasets and compressing them, the resultant compressed file size can be utilized to calculate the compression ratio where

$$\text{Compression ratio} = \frac{\text{compressed size}}{\text{original size}}$$

For paired-end sequenced reads, the sum of the two compressed files is divided by the sum of the two original FASTQ files.

Dataset	**pigz**	**pbzip2**	**fqzcomp**	**LWFQZip**	**fastqz**	**quip**	**compress**
ERR231645	0.33	0.26	0.19	0.18	**0.16**	0.19	0.20
SRR801793	0.36	0.28	0.22	0.2	**0.16**	0.2	0.23
ERR233152	0.28	0.22	0.17	0.17	**0.14**	0.17	0.20
ERR005143	0.29	0.22	0.17	**0.16**	**0.16**	0.17	**0.16**
SRR327342	0.41	0.34	0.25	0.26	**0.21**	0.24	0.28
ERR161544	0.18	0.25	0.19	0.2[3]	**0.16**	0.19	**0.16**
Average	0.31	0.26	0.20	0.20	**0.17**	0.19	0.21

Table 15: Compression ratios of various compression tools on selected datasets

First off, it can be noticed that all the genomics compression tools offer a substantial increase in compression rates than general purpose utilities. Of all the different compression tools, fastqz seems to perform at the top of the list.

However, for our compress tool, the compression file sizes of our new compress tool seems to be significantly higher for 2 datasets - `SRR327342` and `ERR233152`. After looking at the compressed file sizes for the individual components, it is found that the sequences have an unusually high amount of matched tuples per sequenced read. Then, it is found that these two datasets contain an abnormally high number of `N` symbols, which occurs in *every single read.* As most of the other tools seem to discard the `N` symbols, it is an indicator of the our sequence compression algorithm being subpar in this scenario.

As mentioned, our compression tool is specifically targeted towards human genome sequences - at which it is on par with the best compression tools out there in terms of compression ratio. Ignoring the low quality datasets implied from the high density of N symbols, **compress** has the second lowest average compress ratio at 0.18.

7.3.1 Compressed size of dataset components

Furthermore, it would be useful to understand how the compressed file size is distributed between the individual compressed components.

This can be examined by simply reversing the archive stage of the compression process and comparing the filesizes. This allows us to evaluate how well the **compress** processed each component of the FASTQ file and whether the results are close to the theoretical expectations. In figure fig:hsapmakeup, the components of the *H. Sapien* compressed file is visualized. It can be seen that the majority of the compressed filesize is due to the quality scores. This is as expected as mentioned previously the randomness of the quality scores leads to a worse compression ratio compared to the other components.

[3]The reference sequence *hg38* had to be trimmed to 2.1GB for the compression tool *LWFQZIP* to work.

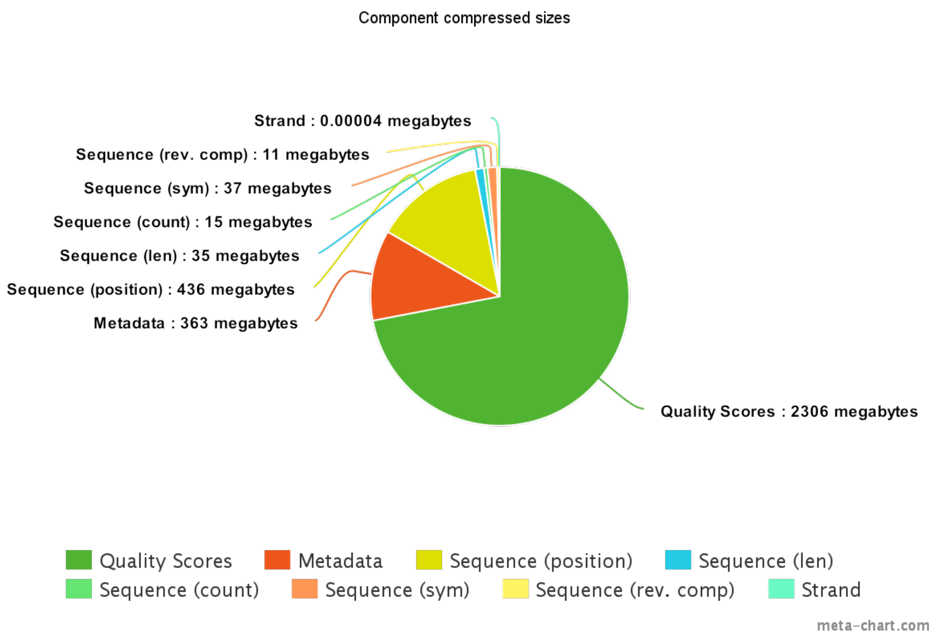


Figure 29: Makeup of compressed *H. Sapien* file.

By examining the components of the other datasets, the results can also corroborate with the previous claim that our compression tool does not compress sequences with low quality (i.e. many unknown nucleobases) well.

File	**Total**	**Metadata**		**Sequence**		**Quality**	
		Size	*%*	*Size*	*%*	*Size*	*%*
ERR161544	3205	363	11.3	535	16.7	2307	72.0
ERR231645	220	30	13.6	44	20.0	146	66.4
SRR801793	342	27	7.9	103	30.1	231	62.0
ERR233152	244	20	8.9	114	46.7	108	44.4
ERR005143	147	22	15.0	36	24.5	89	60.5
SRR327342	1584	50	3.2	603	38.1	931	58.7

Table 16: Makeup of selected compressed FASTQ datasets using **compress**. All file sizes are in megabytes

It can be clearly seen from the table above that for *ERR233152* and *SRR327342*, due to the large amount of unknown symbols in each read, the compressed sequence is more than double the relative size of high quality sequences such as the `ERR165144` dataset.

7.4 Compression Speed

For the compression speed, it should be noted that all the other tools are run with the default settings. Moreover, all the times are averages of three repeated executions to prevent fluctuations in the timings. For the paired-end sequenced reads, the *sum* of the compression time for each of the pairs is taken. Now, the other compression tools in this benchmark are timed and the results listed below. The timing is done via the Linux system wide binary **time** (located in */usr/bin/*).

Then, our compression tool is run with a wide range of settings. Firstly, the CPU version is run using varying number of threads (`N_THREADS`) from 4 to 24 inclusive incremented by 4. Then, the Maxeler dataflow engine implementation of the sequence compression and the normal CPU versions of the rest of the FASTQ components is run with 1, 2, 4, and 8 FPGAs and 16 threads.

The first observation that can be made is that the general purpose compression tools have a faster compression time than all of the genomics compression tools. This is most likely due to the time saved

Dataset	**pigz**	**pbzip2**	**fqzcomp**	**LWFQZip**	**fastqz**	**quip**
ERR231645	11.6	12.1	16.6	355.5	140.9	31.8
SRR801793	37.6	31.1	50.8	1367.9	486.5	85.5
ERR233152	5.2	4.8	8.4	277.5	58.9	18.3
ERR005143	7.6	10.3	12.33	289.3	139.4	32.9
SRR327342	73.7	63.2	102.8	2640.5	930.4	149.2
ERR161544	265.6	358.6	760.1	22320	7523.6	1329

Table 17: Compression speeds of various compression tools on selected datasets

Dataset	**CPU N_THREADS**						**DFE N_DFE**			
	4	*8*	*12*	*16*	*20*	*24*	*1*	*2*	*4*	*8*
ERR231645	62.7	38.9	32.7	32.1	30.3	29.5	36.9	39.9	35.0	33.0
SRR801793	196.4	113.6	91.9	90.3	87.4	83.8	98.7	96.6	88.5	81.2
ERR233152	34.2	21.9	19.1	19.6	19.6	18.4	23.9	23.1	21.3	21.1
ERR005143	68.3	48.4	43.1	45.3	45.7	41.9	42.1	40.3	35.5	38.9
SRR327342	328.4	198.8	207.6	166.4	155.5	153.4	141.5	124.4	115	117.4
ERR161544	2050.5	1164.7	932.4	930.3	882.3	880	690.8	579.3	562.1	540.9

Table 18: Compression speeds of **compress** with different configurations

on not reading in the reference sequence FM-index, which takes substantial time to read (and reconstruct if the bucket size of the FM-index is not 1). The fact that the overhead for the referential compression is significant can be further deduced by the fact that **fqzcomp** has the fastest compression speed out of all the genomics compression tools benchmarked above is coincidentally the only genomics compression tool that *does not* utilize a reference based compression method for the sequences.

Furthermore, the two compression tools (except **compress**) that have the highest compression rates as mentioned in the previous section, i.e. **fastqz** and **LWFQZIP**, are at least one order of magnitude slower than the rest of the compression tools for most datasets.

Focusing on the human genome dataset, **compress**'s dataflow computing implementation is faster than any other genomics compression tool. Even when the CPU version with 8 threads is used, it still ranks second after **fqzcomp**.

However, on the other datasets, **fqzcomp** is 20% to 40% slower than the general purpose compression tools while **compress** (12 threads) and **quip** approximately takes a further twice as long than **fqzcomp** for small datasets (<2.5GB) and 50% longer for medium datasets (2.5GB - 10GB). Even the DFE version does not have a substantial impact when used on smaller datasets as the overall time saved within the sequence compression is not significant in terms of the program's overhead.

7.4.1 Comparing the CPU vs DFE versions of sequence compression

The speedup between the CPU and dataflow versions of **compress** can be measured and evaluated. Instead of measuring the compression of the entire file, the Linux system call `gettimeofday` from `sys/time.h` is called immediately before and after the sequence compression functions for each batch. Then, the times for each batch is summed, which should approximately yield the actual time spent on the sequence compressions for both versions.

The CPU version, DFE with 1 FPGA and DFE with 4 FPGA is compared below:

Firstly, the time it takes to compress the sequences using 4 FPGAs, approaches half of the time it takes for 1 FPGA at larger file sizes. For large files, the more significant the FPGA's acceleration is compared to the CPU. Thirdly, for small files the difference between the CPUs and the FPGAs is less. This is mainly due to the fact that the overhead of the streaming inputs/outputs to and from the dataflow engine becomes significant when the total number of sequences to compress are low.

	CPU	DFE (1 FPGA)	DFE (4 FPGA)
HSap	270.6	91.1	59.5
LPneumo	20.4	14.6	9.4
Ecoli	11.3	7.4	5.3
PSyr	5.7	3.4	2.5
PAe	7.5	9.3	6.7
SC	79.3	60.5	30.3

Table 19: Time spent on Sequence Compression (in seconds)

7.5 Decompression Speed

Dataset	pigz	pbzip2	fqzcomp	LWFQZip	fastqz	quip
ERR231645	9.2	5.4	18.9	32.6	158.1	49.9
SRR801793	23.7	17	68.5	91.7	525.9	196.1
ERR233152	5	2.3	11.7	18.8	63.2	29.9
ERR005143	7.5	4.2	17.5	30.5	162.7	43.4
SRR327342	52.1	30.2	117.8	175.3	985.5	256.4
ERR161544	139.1	272.9	816.2	13276.8	7699.4	2070.5

Table 20: Decompression speeds of various compression tools on selected datasets

The benchmarking of the decompression speeds are done similarly to that of the compression speed.

Dataset	CPU N_THREADS					
	4	*8*	*12*	*16*	*20*	*24*
ERR231645	48.5	49.6	46.3	49.5	46.7	47.1
SRR801793	148.8	146.6	146.8	145.6	145.7	145.3
ERR233152	17.8	19.5	19.6	20.5	19.5	19.5
ERR005143	35.5	35.6	38	39.5	35.2	35.7
SRR327342	270.2	269.3	269.8	267.2	268.7	269.5 7
ERR161544	1551.6	1554.2	1533.4	1469.8	1440	1438.1

Table 21: Decompression speeds of **compress** with different configurations

Once again, the general purpose compression tools are substantially faster than all the genomics data compression tools. The relative decompression speeds of all the compression tools are very similar to the compression speeds. From the benchmarks it can be seen that **LWFQZIP** and **fastqz** is still an order of magnitude worse.

However, due to the fact that the quality scores decompression in **compress** has not and cannot be parallelized, the quality scores decompression seem to be the limiting factor to the performance of **compress**'s decompression speeds, and thus varying the amount of threads used does not seem to have any significant improvement on the decompression speeds unless doing so far a large dataset such as the human genome. Nonetheless, it still is faster than all the genomics compression tools except **fqzcomp**. Nonetheless, **compress** is still faster than all the algorithms that employ referential compression for sequences.

7.6 Maximum Memory Usage

It is very difficult to compare the maximum memory usage as the disparity between the the reference-based compression tools which require the complete reference genome or FM-index to be loaded into memory and the other non-reference-based compression tools cannot be compared. Furthermore, due to the use of multithreading of most tools, the machines on which the compression tools are running should not be running other processes require high amounts of processing time or memory use. Thus, the evaluation of the memory usage may not necessarily require the comparison of different compression tools using a scale where the least memory used is the best. Rather, only **compress** is required to be evaluated by checking whether the maximum memory use is less than the memory found commonly on computers.

	CPU	**DFE**
H Sapien	22.0	25.2
E Coli	8.8	10.3
L Pneumophila	6.4	8.2
P Aeruginosa	5.4	6.8
P Syringae	6.3	8.0
S Cerevisiae	12.2	14.5

Table 22: Maximum memory usage (in gigabytes) of **compress** on selected datasets

The dataflow computing implementations require significantly higher amounts of memory, mostly used for the buffers for communication between the CPU running the main thread and the DFE running the sequence compression part. The DFE version allocates a block of memory that can store the entire compressed sequence output for a single batch which is written to by the DFE. On the other hand, the CPU version has a much smaller buffer which is flushed to file when the buffer is full. This is the main difference in the memory usage of the two versions of **compress**.

Moreover, using the 16GB goal set in the initial aims of the compression tool, it can be found that for all of the selected sequences, all of them except **H. Sapiens** can fit inside 16GB.

It should be noted that it is possible to decrease the memory usage at the expense of compression and decompression time. In these benchmarks, the suffix array used has not been sampled. Thus, the human genome reference `hg38.fmt` generates a corresponding suffix array of approximately 5.5 gigabytes. To not surpass the available memory, researchers and other users should sample the suffix array according to their system's available memory. If the suffix array is sampled and stored for every d elements, there will be a corresponding d-fold decrease in the file size of the suffix array.

7.7 Summary

Firstly, the main goal of the compression tools, to achieve a high degree of compression, can be seen from the values of the compression ratio. The compression ratio of our compression tool, the fastest compression tool **pbzip2**, the fastest genomics compression tool **fqzcomp**, and the compression tool with the highest compression rate **fastqz** is plot in the figure below.

compress performs worse relative to its peers when used on low quality sequences populated with unknown bases. However, for high quality sequences will no unknown bases, its compression ratio is mostly on par with the best compression tools.

Additionally, a good metric to depict the compression speed would be to normalize the speed by dividing the compression time by the file size. Four tools are selected for comparison - the fastest compression tool, which is **pbzip2**, the compression tool with the highest compression ratios, **fqzcomp**, and then **compress**'s CPU version with 12 threads and the dataflow engine version.

From the figure above it can be seen that while the CPU version of **compress** is faster for small files (<2.5GB), the dataflow computing version becomes significantly faster at compressing bigger files.

Using the results from the benchmarks, there are a few observations that can be tentatively made:

- General purpose compression utilities **pbzip2** and **pigz** - fastest compression and decompression speeds, low memory usage due to no referential compression, but the compressed file sizes can be up to double the best genomics compression tools
- **fqzcomp** - Without the overhead reference, it performs well in the compression speed, decompression speed and memory usage benchmarks, but have the worst compression ratio out of the

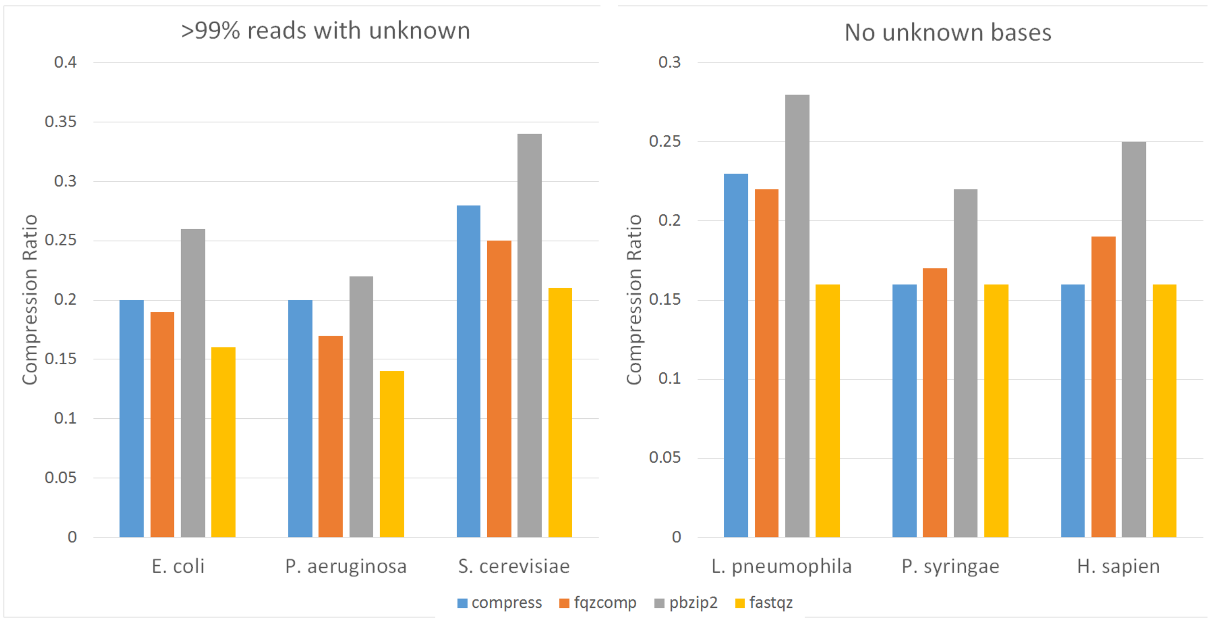


Figure 30: Compression ratio of **compress**, **fqzcomp**, **pbzip2** and **fastqz**

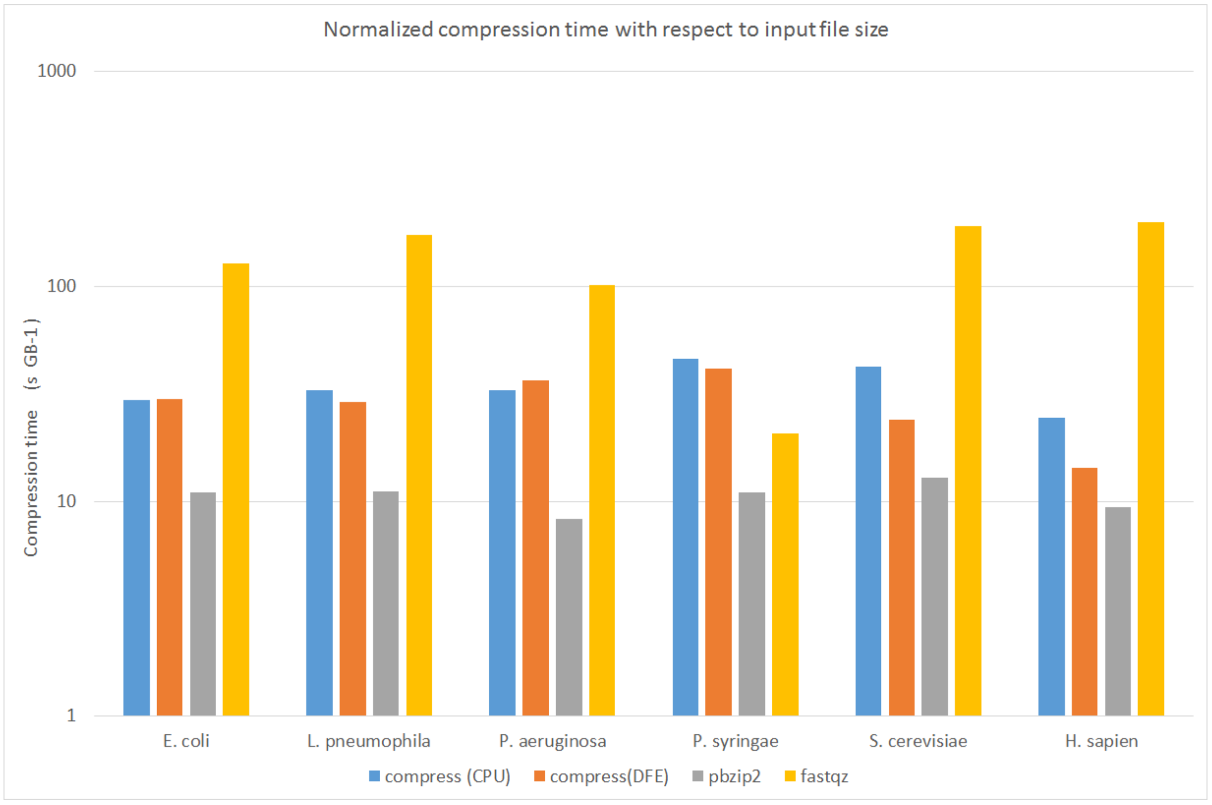


Figure 31: Normalized compression times for selected compression tools. Note that the vertical axis is in logarithmic scale.

genomics compression tools. The benchmark results suggest that its performance in every area is between the other genomic compression tools and the general purpose compression tools.

- **LWFQZip** - an order of magnitude slower than *all* the other compression tools on many of the sampled datasets.
- **fastqz** - the second slowest out of all the compression tools, but consistently yields the best compression results.
- **quip** - the version with assembly enabled is significantly slower with little improvement in compression ratio. Without its de novo assembly, it is above average in all the benchmarks.
- **compress**
  - Given high quality datasets without unknown characters, it is on par with the best compression rates.
  - The larger the dataset, the faster the compression and decompression speed is relative to its peers and for a dataset as large as the human genome, it can surpass the compression speed of **fqzcomp** (but is still behind the general purpose compression tools)
  - With only 12 threads on the CPU version, it achieves better or similar compression speeds to all the algorithms with reference-based sequence compression.

Note that due to faulty assumptions by **fastqz** and **fqzcomp** which do not allow repeated sequence identifiers/metadata in the strand, all the FASTQ datasets with the exception of the *H. Sapien* has been modified to remove all repetitions of the metadata from the strand. As mentioned previously, **compress** attempts to make as few assumptions about the FASTQ data as possible to ensure that it works correctly for the overwhelming large majority of FASTQ and FASTA sequence files.

8 Conclusion

With the advances next-generation sequencing technology in the past decade, the available amount of genomic sequence data has exponentially increased. This poses the problem that the throughput of next-generation sequencing machines has increased at a faster rate than the increase in the capacity of memory storage as predicted by Moore's law. To prevent the storage of genomic sequence data from becoming the bottleneck of genomics research in the near future, this report proposes the adaption of a new compression algorithm to alleviate the problem of data storage. The output data from next-generation sequencing machines, specifically the high quality human genomes, is analyzed to find optimal strategies in the compression of each of the components. Furthermore, this algorithm is accelerated via the usage of multithreading computing paradigms and dataflow computing engines. Results indicate that the the compression ratio of human genomes is equal to that of the best genomic compression tools while the compression time is decreased by up to twenty-fold.

As mentioned in the introduction, the contributions this report has made is reiterated:

- **C1** - We develop a novel lossless compression tool for genomic data in the FASTA and FASTQ formats. The compression tool yields up to a 6-fold decrease in file size, and is especially effective given high quality, high coverage and long sequences such as the human genome. We develop novel methods to compress the metadata (also known as the sequence identifiers) and the quality scores:
  - **C1.1** Firstly, the metadata is tokenized. Then, these tokens are analyzed and classified into archetypes with common behaviour. Depending on the type, the token is then encoded into minimal amounts of memory using a highly-adaptive novel algorithm to find the optimal encoding for each field based on the characteristics of the archetype and the values of the input metadata. The compression of the metadata achieves greater than a 10-fold decrease in file size.
  - **C1.2** The method to compress quality scores involves initially the quality scores are translated into run-length encoding symbols. The quality scores are then grouped in order to reduce the entropy, and the finally compressed via arithmetic encoding;
- **C2** - We optimize the compression tool by acceleration using novel greedy algorithms in determining the encoding of the quality scores and metadata, multithreading techniques and OpenMP to improve the performance in terms of compression and decompression speeds. The sequence compression is further accelerated using Maxeler's dataflow computing engine. The acceleration reduces the overall compression time of a complete human genome (approx. 19 gigabytes) to 270 seconds, reduced from over 7000 seconds which is required by compression tools that yield the same compression ratio;
- **C3** - We employ various techniques to reduce the maximum memory usage, such as subdividing the input file to be compressed and reducing the size of the reference index and suffix array in the reference-based sequence compression stage by sampling during precomputation of the references; and
- **C4** - The compression tool presented is evaluated by benchmarking against other existing compression tools. Six FASTQ datasets are chosen, containing various species, read lengths and read counts. The six compression utilities, of which two are general purpose compression utilities and the other four domain-specific compression tools designed for FASTQ data, is then benchmarked using the following criterion: (1) compression rate, (2) compression speed, and (3) decompression speed.

8.1 Achievement

From a more quantitative standpoint, we can evaluate the compression tool in terms of the concrete aims laid out at the beginning of this report.

- **A1** *Compression ratio* - the compression ratio is substantially better than general purpose utilities. Furthermore, for high coverage and high quality sequences without unknown symbols, the compression rates can reach at least 0.16, which is equal or better to the most commonly used genomics compression tools.

- **A2** *Compression and decompression speed* - the compression speed has been vastly accelerated due to the parallelization applied to most of the compression and decompression pipeline. Moreover, the compression speed large files is an order of magnitude faster than the compression tools that can produce the same compression ratio. Furthermore, the majority of the algorithm has been modified to run in parallel without any significant worsening in compression ratio. Parts of the compression algorithm has also been accelerated via the use of dataflow computing engines, which enables the compression and decompression speed for medium to large FASTQ files (above 2.5GB) to be faster than all the other common genomics compression tools. **pbzip2** is 33% faster than **compress** for large files, which is at the expense of the compressed file having an increased size of 40%.
- **A3** *Memory usage* - the memory usage might be higher that desired for large files in the CPU version of **compress** as the reference FM-index is loaded into RAM. However, for small files, the memory usage is lower than the quantitative goal set forth in the aims. Moreover, it should be noted that systems designed for genomics analysis usually have large amounts of memory installed, which allows **compress** to be used by the targeted users without incurring huge performance penalties. Further reduction in memory usage is possible if necessary through the sampling of suffix array used in the referential compression stage.

It can be concluded that the compression tool described above achieves most if not all of the intial objectives. On the main focus of compress, which is the compression of human genome sequences, the benchmarks show that **compress** is tied with the best compression ratio and has faster compression speed than all other genomics compression tools that we have tested.

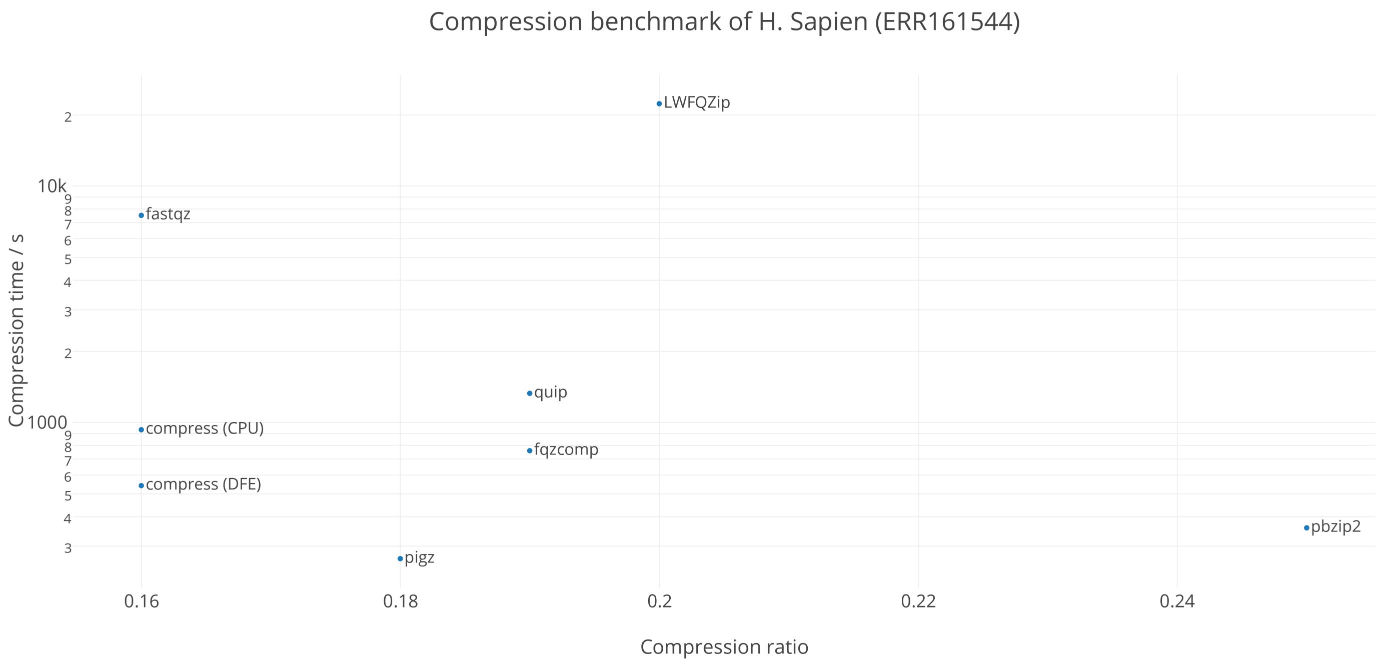


Figure 32: Compression benchmarks of H. Sapien dataset

Using the figure above, it can be seen that the optimal choices for the compression of high quality sequences would either be **pigz** or **compress**. The difference between the two options is once again the frequently mentioned tradeoff between compression time and compression ratio. Currently, despite the unparalleled growth of sequencing technology, the speed of next generation sequencing has yed to catch up with the time it takes for **compress** to process the same file. Thus, considering the implication that the present bottleneck is not the compression time, the 10% further reduction in compressed file size relative to **pigz** may lead to the potential selection of **compress** during the compression of FASTQ and FASTA data for genomics research.

8.2 Future Work

In this section, possible work that is based on or can extend the compression algorithm described in this project is presented.

8.2.1 Further improvements for *compress*

Although our compression tool has demonstrated its effective compression ratio, there is still much room for improvement. Four possible improvements are described below.

8.2.1.A Dealing with unknown nucleobases

Currently, the method dealing with unknown nucleobases is highly inefficient. While if the unknown bases are at the end, the compression ratio will not be severely affected, nonetheless if the unknown bases are in the middle of a seqeuenced read the mapped sequence will have to split into two tuples in addition to the tuple encoding the unknown base itself. This can cause a huge bloat in the compressed file size if the input FASTQ file contains low quality sequences. Although the newer sequencing machines rarely output unknown symbols, to allow the compression tool to be retroactively applied to the comparatively older data, this problem must be tackled. If a suitable treatment of unknown bases is implemented, **compress** may have one of the best compression ratios regardless of the quality of the input FASTQ datasets.

8.2.1.B Parallelization of arithmetic decoding

By definition, arithmetic encoding, along with most variable length codes (a notable exception being unary codes) cannot be decompressed in parallel as there is no way of discerning whether a bit denotes the beginning of a codeword or not. However, there are proposed modifications to the arithmetic coding that allows an increased compression and decompression speed. One such proposal is described in the paper *Parallel implementation of a multialphabet arithmetic coding algorithm.*[18]

8.2.1.C Further exploration of quality scores modelling

In this project, a feedforward neural network was trained in an attempt to find a hidden correlation between successive quality scores. Nonetheless, the architecture of a feedforward neural network with one layer may not be sufficient to model the evolution of quality score characters. Thus, more complex neural networks may be trained and applied to predict successive quality scores efficiently.

8.2.1.D Dataflow implementation of metadata and quality scores compression

For relatively larger datasets, the dataflow computing acceleration becomes increasingly significant in terms of the compression and decompression time. Most of the algorithmic processes utilized in **compress** should not be difficult to implement in dataflow computing engines, with the exception of the bit-emitting process during arithmetic encoding which may be troublesome.

8.2.2 Human Genome Research

There is a potential impact of a high speed, high throughput and effective compression tool on research revolving around human genomes or other high-quality or large sequences. Multi-disciplinary team efforts play a pivotal role in advancing the frontier of knowledge. This is especially true for computer science and engineering, which has been the cause of advances in technology across a multitude of scientific fields. From the perspective of genomic research, high compression rates facilitates the communication and transfer of genomic data, while high compression and decompression speeds allow sequencing data to be immediately stored and quickly retrieved at a later date.

8.3 Last Remarks

Devising suitable methods for the compression of data as complex as DNA sequences is a challenging task. We have attempted to develop a method that although works well on a variety of sampled data, can undoubtedly undergo future improvement in terms of both compression speed and compression ratio. We have successfully implemented and accelerated a compression tool which allows sequencing data to be stored and transferred easily.

The research on human genomics has barely scratched the surface so far. Debilitating genetic medical conditions ranging from cancer and Alzheimer's to Huntington's disease that invoke pain and suffering in the patients and their family and friends could be cured and eradicated with genomic research in the future, similar to how vaccines have lead to the eradication of smallpox in the last century.

An inter-disciplinary effort will accelerate the development of treatments. The application of chemical processes such as ligation coupled with computer engineered machines has automated the DNA sequencing processes and demonstrated how science and engineering complements each other. This project has developed a novel compression utility which after evaluation shows that it can be potentially useful for genetic researchers, and displays the ability of concepts in information theory and computer science to be applied in other scientific disciplines.

References

[1] James Arram, *FPGA Acceleration of Reference-Based Compression for Genomic Data*, Imperial College London, 2015.

[2] JK Bonfield, *Compression of FASTQ and SAM format sequencing data*, Wellcome Trust Sanger Institute, Cambridge, United Kingdom, 2015.

[3] M. Burrows and D. Wheeler. *A block-sorting lossless data compression algorithm.* Technical report, Digital Equipment Corporation, 1994.

[4] A. Chacon et al. *n-step FM-Index for Faster Pattern Matching.* ´ Procedia Computer Science, 18(0):70 – 79, 2013

[5] *Mapping and Sequencing the Human Genome*, Committee on Mapping and Sequencing the Human Genome, National Research Council. 1988. ISBN: 978-0-309-07462-9

[6] Sara El-Metwally, Osama M. Ouda, Mohamed Helmy. *New Horizons in Next-Generation Sequencing. Next Generation Sequencing Technologies and Challenges in Sequence Assembly*, SpringerBriefs in Systems Biology Volume 7, 2014, pp 51-59.

[7] *ERR005143 Whole Genome Sequencing of Pseudomonas syringae.* Available at http://sra.dnanexus.com/runs/ERR005143/studies

[8] *ERR161544 Whole Genome Sequencing of Homo sapiens.* Available at http://sra.dnanexus.com/runs/ERR161544

[9] *ERR231645 Whole Genome Sequencing of Escherichia coli.* Available at https://trace.ddbj.nig.ac.jp/DRASearch/run?acc=ERR231645

[10] *ERR233152 Whole Genome Sequencing of Pseudomonas Aeruginosa.* Available at http://www.ncbi.nlm.nih.gov/sra/?term=ERR233152

[11] *FASTQ Files - CASAVA v1.8.2 DNA Sequencing ANalaysis Workflow*, Illumina, Inc. Available at http://support.illumina.com/help/SequencingAnalysisWorkflow/Content/Vault/Informatics/Sequencing_Analysis/CASAVA/swSEQ_mCA_FASTQFiles.htm0

[12] *Funding History*, National Human Genome Research Institute Available at https://www.genome.gov/pages/about/budget/fundinghistory90-11.pdf

[13] Jeff Gilchrist, *Parallel BZIP2 (PBZIP2) Data Compression Software.* Available at http://compression.ca/pbzip2/

[14] Paul G. Howard, Jeffrey S. Vitter; *Arithmetic coding for data compression*, Proceedings of the IEEE 82 (6): 857–865, 1994.doi:10.1109/5.286189.

[15] *Illumina, Inc. Common Stock Quote*, NASDAQ. Available at `http://www.nasdaq.com/symbol/ilmn`. Accessed at Jun 13, 2016.

[16] Daniel C. Jones, Walter L. Ruzzo, Xinxia Peng, and Michael G. Katze, *Compression of next-generation sequencing reads aided by highly efficient de novo assembly*, Nucleic Acids Res. 2012 : gks754v1-gks754.

[17] Peter A. Jones et al. *The Epigenomics of Cancer.* Cell, Volume 128, Issue 4, p683 - 692.

[18] S Mahapatra, J L Nunez, C Feregrino-Uribe and S Jones, *Parallel implementation of a multialphabet arithmetic coding algorithm*, Lougborough University.

[19] *Paired-end sequencing*, Illumina Technologies. http://www.illumina.com/technology/next-generation-sequencing/paired-end-sequencing_assay.html

[20] O. Pell, V. Averbukh, *Maximum Performance Computing with Dataflow Engines.* Maxeler Technologies Computing in Science & Engineering, July-Aug. 2012, doi: 10.1109/MCSE.2012.78.

[21] Michael A Quail, Miriam Smith, Paul Coupland, Thomas D Otto, Simon R Harris, Thomas R Connor, Anna Bertoni, Harold P Swerdlow, and Yong Gu, *A tale of three next generation*

sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers, BMC Genomics. 2012; 13: 341.

[22] Michael J Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Inc. 2004. ISBN 0-07-058201-7

[23] F Sanger, AR Coulson (May 1975). *A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase*, Journal of Molecular Biology 94.

[24] C.E. Shannon, *A Mathematical Theory of Communication.* Bell System Technical Journal, vol. 27, pp. 379–423, 623-656, July, October, 1948

[25] Mamta Sharma, *Compression Using Huffman Coding.* IJCSNS International Journal of Computer Science and Network Security, VOL.10 No.5, May 2010 133

[26] *SOLiD system sequencing*, Applied Biosciences. Available at: http://www3.appliedbiosystems.com/cms/groups/mcb_marketing/documents/generaldocuments/cms_061241.pdf Accessed May 15, 2016.

[27] *SRR327342 Genome Sequencing of Saccharomyces cerevisiae l14.* Available at http://sra.dnanexus.com/runs/SRR327342

[28] *SRR801793 General Sample for Legionella pneumophila.* Available at http://sra.dnanexus.com/runs/SRR801793/samples

[29] T.P. Vogl, J.K. Mangis, A.K. Rigler, W.T. Zink, and D.L. Alkon, *Accelerating the convergence of the backpropagation method," Biological Cybernetics*, Vol. 59, 1988, pp. 257–263

[30] K A Wetterstrand. *DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)* Available at: www.genome.gov/sequencingcostsdata.

[31] Yang Zhang, *FASTA format*, University of Michigan. Available at http://zhanglab.ccmb.med.umich.edu/FASTA

[32] Zhang Y, Li L, Yang Y, Yang X, He S, and Zhu Z, *Light-weight reference-based compression of FASTQ data*, 2015.

Appendices

A Quality score frequencies

!	**"**	#	**$**	%	&	**’**	(	
0	0	1567228	0	8877	9156	19829	26096	
)	*	+	,	**-**	**.**	/	**0**	
23180	34243	47254	52308	39501	90169	55292	79515	
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	
81286	102120	122755	160220	213786	202781	232397	211097	
9	**:**	;	<	=	>	**?**	**@**	
337662	383548	409741	608926	537489	769120	1108534	1119625	
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**
1283933	2617060	2476649	4132378	7464186	12214754	8363603	52770095	23607

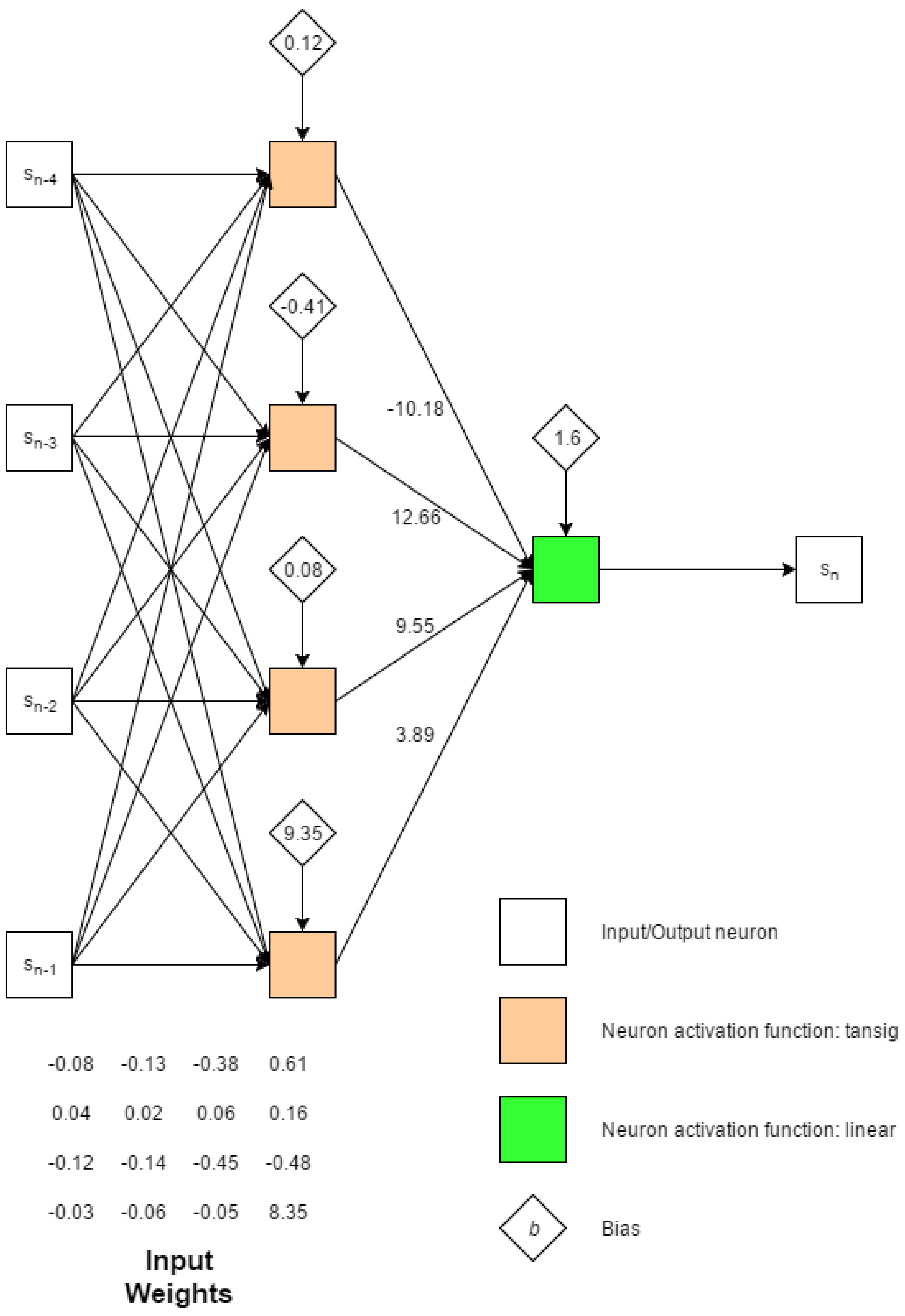
Table 23: Quality score frequencies (first 100 million in *ERR161544.fastq*)

Prev Next	*C*	*D*	*E*	*F*	*G*	*H*	*I*
C	293739	289828	331823	396294	148652	274040	584
D	273691	686925	541446	655172	327138	568007	1195
E	315997	516712	1654220	1320392	748154	1621599	2802
F	390265	621339	1261107	3105863	1246977	4279822	3927
G	147758	320969	744487	1227258	2854626	2449881	9087
H	322487	646156	1743506	4405630	2444751	42198798	2236
I	558	1062	2794	3662	8866	3368	1214

Table 24: Excerpt of successive quality score frequencies[4]

[4]As the original table is 41-by-41, it cannot be fit into the page. Thus, the section with the highest quality scores (which, coincidentally, contains the highest frequencies as well) is chosen.

B Neural network diagram



0.12
-0.41
0.08
9.35
1.6
s_{n-4}
s_{n-3}
s_{n-2}
s_{n-1}
s_n
-10.18
12.66
9.55
3.89
-0.08 -0.13 -0.38 0.61
0.04 0.02 0.06 0.16
-0.12 -0.14 -0.45 -0.48
-0.03 -0.06 -0.05 8.35
Input Weights
Input/Output neuron
Neuron activation function: tansig
Neuron activation function: linear
b
Bias