Imperial College London
Departement of Computing

Independent Study Option

Optimised Compression of Genetic Sequencing Data

as part of the course
MSc Advanced Computing

Moritz Pflanzer

29th April 2015

Supervisor
Wayne Luk

Abstract

The aim of this report is to outline potential applications of hardware acceleration for the purpose of genetic sequencing data compression. First a short summary of the development in the field of DNA sequencing is provided to explain the need for efficient compression tools. It is followed by an overview over the basic genetic structures which can be exploited to design efficient and specialised compressors. The main part of the report comprises a comprehensive description, evaluation and comparison of the algorithms currently considered as state of the art. The focus of the evaluation is not only on the compression ratio but also on the throughput achieved for compressing the data and (especially) for decompression. The most efficient algorithms are then considered for an adaptation to Dataflow computing to benefit from the possibly high throughput and low energy consumption on FPGAs. The scope of this report is limited to referring to existing hardware implementations that can be used as building blocks for fully featured compressors.

Contents

List of Figures

List of Tables

List of Codes

1

Introduction

The aim of genetics can be formulated as to understand the relation between genotypes and phenotypes [Mar08]. This knowledge is important for a better grasp of the genetic diversity among the existing species [Now10] and to gain deeper information about the mutational spectrum of (model) organisms [Mar08]. Ultimately, this information leads to both global scale improvements of the public health [GW13] and small scale *personal genomics* [Ans09] and personalised medicines [Now10]. Beside the active research in analysing human cancers [Rei09; PS08] understanding the impact of different genotypes can also help with the research about mental diseases, like schizophrenia or autism, and the impact of microbial life [Ans09].

The essentiality of a deep understanding of the meaning of the DNA has driven the need for analysis methods early on. Already in the mid to late 1970s Sanger et al. [SC75] and Maxam et al. [MG77] developed independently of each other a method to sequence DNA to determine the precise order of nucleotides contained. In the beginning the method developed by Maxam et al. was more efficient but Sanger et al. refined their technique in the following years and the resulting *dideoxy method* became the most used method for DNA sequencing for the next 30 years [Hut07]. During this time the read length increased from initially 15–200 nucleotides per fragment [SNC77] to up to 1 000 bases [Now10]. In contrast, the costs for sequencing have decreased exponentially over the time [Mar08]. While it took 13 years and approximately $3 billion to obtain the first complete sequence of the human genome ("Human Genome Project") using new techniques and automated processes it is now possible to get a "10-fold coverage of the human genome (30 Gb DNA sequence) [...] in a single run for no more than $15 000–$20 000" [Rei09]. Moreover, instruments have been announced which will be able to sequence the whole human genome in less than a day for under $1 000.

These new *Next-Generation Sequencing* (NGS) technologies are different from the automated Sanger sequencing in that they are able to perform millions of sequencing reactions in parallel thus allowing a much higher throughput. Since NGS results in much shorter reads than obtained by Sanger sequencing their major field of application is genome *resequencing*. Even relatively short reads can be mapped onto an existing reference sequence with high confidentiality. For the more complex *de novo* sequencing and assembly NGS have first been combined with Sanger sequencing but due to that fact that the read lengths are getting competitive with those of the Sanger sequencing and improved assembly algorithms the use on their own gets increasingly attractive [Now10].

The rapid and huge enhancements in the area of sequencing technologies come along with new challenges in the field of bioinformatics. In the first place there is need for fast algorithms to solve the *fragment assembly* problem. Most algorithms are based on reductions to variations of the *Eulerian Circuit Problem* (ECP) or similar graph problems [PS11, Chapter 3]. Further, the enormous amount of data produced by the NGS has not only to be processed but also do the results of the sequencing have to be stored and shared between institutions. In recent years the growth in raw output data generated by the NGS platforms has exceeded *Moore's Law* and thus in future more data will be produced than the amount which can be analysed and stored efficiently [Kah11]. Thus, although storage and bandwidth costs are steadily decreasing at a rate of 30 %–40 % per year [Mee14, Slides 71 and 72] the costs for maintaining DNA sequence archives have not stopped to grow [GRU14; DG13]. The reduced storage and transportation costs might be able to compensate the increasing amount of data that gets generated but due to the lower costs of running NGS platforms more and more institutions can afford to perform experiments.

The two most popular data formats for storing sequencing results are *FASTA/FASTQ* and *SAM/BAM*. Aside from the BAM format they are entirely based on an ASCII plain text representation of the base calls and quality scores for each sequenced read. Now, to reduce the archival costs, one can compress these formats – which has been done with BAM – or come up with new formats that exploit the characteristics of the DNA and also the sequencing meta-data to allow for higher compression rates. Therefore Chapter 2 introduces the basic structural concepts of genetic information and provides a more detailed view on the existing file formats.

In Chapter 3 the different approaches how to compress the data are outlined and compared against each other. From this key assumptions are derived which allow to focus on specific subgroups of algorithms in Chapter 4. First the algorithms are presented and the main features are outlined. Then in Chapter 5 the algorithms are all tested on the same test data. As the authors of the different papers use different data or metrics simply comparing

the claimed results is not possible. Finally, in Chapter 6 the results are discussed and further improvements and research directions are stated.

2

Theoretical Background

In this chapter the basic structure of the genetic information is described as it can be used to design more effective compression algorithms. Moreover, the specifications of the existing sequencing data file formats are outlined as they indicate which features the compressed formats should comprise.

2.1 Structure of Genetic Information

The entire genetic material is described by the term *genome* which consists of different *genes*. Genes divide the encoded information into logical groups and are mapped to specific *loci* on a *chromosome* inside the cell. The chromosomes are built out of DNA molecules arranged in a two stranded double helix (Figure 2.1). The two strands are connected by *nucleotide* pairs where each base is drawn from the four letter alphabet `{A, C, G, T}`. The strands are said to be antiparallel in the sense that one is the reverse of the other and since the bases normally form the pairs `A`—`T` and `C`—`G` the strands are also complementary. The direction of a strand is determined by the position of the sugar of each nucleotide which can either be $5'$ or $3'$. If not specified otherwise a left-to-right orientation is assumed which corresponds to the $5'$ to $3'$ polarity.

To encode the alphabet of the 20 naturally occurring *amino acids* the nucleotides are grouped into *codons* with a length of 3 letters. As there are $4^3 = 64$ trinucleotides possible each amino acid is encoded by more than one codon (Appendix A.1). The triplets are not uniformly distributed over all amino acids and 3 of the 64 codons are used as special *stop codons* that do not encode any amino acid.

In general not all the information contained in the DNA is translated into gene products. Only *Exons* form the *coding* regions of the DNA that are needed to built RNA or proteins. In contrast *Introns* are *non-coding*

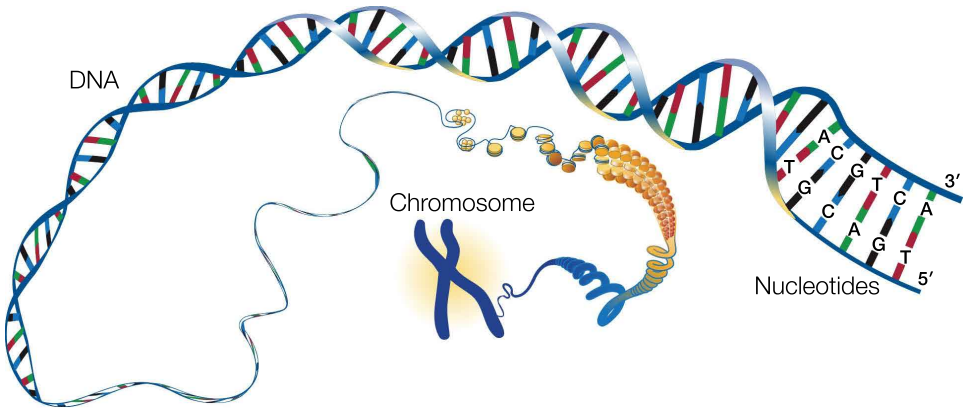


Figure 2.1 – Structure of genetic information. The genetic information is encoded through nucleotide base pairs which are arranged in a two stranded double helix. This DNA forms the chromosomes. The graphic is adapted from *DNA Testing Expert* (`http://dnatestingexpert.com/home-2/amazing-dna/`).

segments which are removed during the translation. Therefore their main purpose is to separate different *Exons*. These spacer regions consist mainly of *microsatellites* which are short, highly repetitive k-mers. In addition to a varying number of repetitions genomes of the same species differ by *single-nucleotide polymorphisms* (SNP), *insertions* and *deletions* (INDEL). A SNP describes the mutation of a single base whereas INDELs are either the occurrence of an extra base in the sequence or the lack of a base.

More detailed information about the biological procedures involved in the encoding and transcription can be found in the book *Computational Genome Analysis* from Deonier et al. [DTW05]. It is excluded from this report as it seems not relevant for the topic of genetic sequencing data compression.

2.2 Sequencing Data Storage Formats

The most commonly used formats to store DNA sequencing information are the *FASTA/FASTQ* and *SAM/BAM* formats [How13]. Aside from the BAM format all formats are plain text ASCII formats designed in such a way that they are easily human readable. The BAM format is a *BGZP* compressed version of the SAM format. This compression is a blocked version of the standard *gzip* format but allows additionally for random access after an index has been built.

The FASTA format This is the simplest one of the four formats. It just contains optional meta-information and the DNA reads (Figure 2.2). The sequences are encoded as strings of either nucleotide acids or proteins. Nucleotides are encoded using the four letters `{A, C, G, T}` plus additional

```
@SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
+SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36
IIIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

Figure 2.2 – Example of a FASTQ formatted DNA sequence of *Escherichia coli*. The first and the third line contain meta-information about the DNA sequence and the experiment in which is has been obtained. The second line is the actual DNA sequence encoded as nucleotide symbols. The fourth line contains the Phred quality scores encoded as ASCII characters. To retrieve a valid FASTA formatted file the third and the fourth line have to be omitted.

letters for positions where the sequencing was not accurate and a hyphen to indicate a gap. The encoding of proteins uses the letter representations from the DNA or RNA codon table (Appendix A.1) plus the additional symbols `*` for a stop position, `X` for undetermined protein and a hyphen for a gap. The format allows to store multiple sequences/reads in one file. The different sequences are separated by the line containing meta-information and are therefore allowed to span multiple lines [NCB07].

The FASTQ format In addition to the raw DNA sequences the FASTQ format contains a quality value, i.e. the probability that the corresponding base is incorrect, for each nucleotide (Figure 2.2). Thus, unlike the FASTA format, the FASTQ format only allows nucleotides to be stored and not codons and the sequences are not allowed to be wrapped over multiple lines. The probabilities are first converted into a *Phred quality score* $Q_{\text{Phred}} = -10\log_{10}(p)$ which is then encoded as printable ASCII character by adding 33. The highest score that can be stored is 93 but raw read data rarely exceeds a score of 60. The FASTQ format is fully backwards compatible with the FASTA format such that FASTQ formatted data can be read out as FASTA formatted data [Coc+10; Ill11].

The SAM format In contrast to the FAST* sequence formats the SAM format does not simple store the sequences of nucleotide symbols but aligns shorter segments (of nucleotide symbols) according to a reference sequence (Figure 2.3). The relation between each such segment and the reference sequence (CIGAR string) is also stored. As in the FASTQ format each nucleotide is assigned a quality value encoded as Phred quality score [SFS15b].

The BAM format To compensate the larger file size of the SAM format compared to FAST* formatted files the BAM format specifies how to compress SAM formatted files. The key feature is to still enable random access to the compressed sequences. For that purpose the *BGZF* compression is introduced. It is based on the standard *gzip* compression algorithms and

```
@HD VN:1.5 SO:coordinate
@SQ SN:ref LN:45
r001    99 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGATACTG *
r002     0 ref  9 30 3S6M1P1I4M *  0   0 AAAAGATAAGGATA     *
r003     0 ref  9 30 5S6M       *  0   0 GCCTAAGCTAA        *
r004     0 ref 16 30 6M14N5M    *  0   0 ATAGCTTCAGC        *
r003  2064 ref 29 17 6H5M       *  0   0 TAGGC              *
r001   147 ref 37 30 9M         =  7 -39 CAGCGGCAT          *
```

Figure 2.3 – Example of a SAM formatted DNA sequence. The first two line contain meta-information about the DNA material. All other lines correspond to segment which have been aligned to the reference sequence. The fourth column describes the start position of the segment in the reference sequence, the fifth column contains the Phred quality scores. The sixth column describes the operations that have been performed to align the segment (tenth column) and the reference. The example is adapted from [SFS15b].

compresses the SAM formatted files blockwise. The random access is granted by generating an index file which contains virtual offsets into the compressed sequence. Decompression is possible by just using the *gunzip* tool. As the normal compression algorithms, namely *gzip* and *bzip2*, do not achieve very high compression rates on DNA sequence data the new format *CRAM* has been introduced which allows to replace the compression algorithms with specialised variants [SFS15b; SFS15a].

3

Compression Techniques

When talking about compression of DNA sequences it has to be assumed that the information contained in the DNA is not stored in a truly random way. Otherwise the best compression that could be achieved would be 2 bit per base as at least the four symbols `{A, C, G, T}` have to be encoded. However, this assumption can safely be made since the DNA is used as a blueprint to construct proteins in living organisms [BS13]. Having only random information this seems to to impossible.

A simple first step to achieve compression of genetic sequencing data would be to use general purpose compression tools like *compress*, *gzip* or *bzip2*. These tools are based on the well-known and simple *Lempel-Ziv(-Welch)* (LZ/LZW) algorithm and the *Burrows-Wheeler transform* (BWT) combined with a *Move-to-Front* (MTF) compression. Additionally the results are improved by applying *Huffman coding* as a final step. Compared to the conventionally used file storage formats (see Section 2.2) these algorithms result in smaller file sizes. But they are hardly able to perform better than the theoretical limit of 2 bit per base which can be reached by just encoding the nucleotides efficiently [BS13]. This is due to fact that these general purpose algorithms do not exploit the specific structures of DNA. For instance the BWT is in principle not unsuited for DNA compression and is used in various compressors, e.g. [JSC14; EV14; Cro+15]. One of the reasons why it does not work for the *bzip2* algorithm is that the window size might be to small [MSI00]. Considering the structure of DNA the repetitive patterns occur all over the genome and not just locally. Therefore, to achieve good results for example a large window size is needed.

3.1 Specialised Compression Approaches

Compression algorithms explicitly designed to achieve a high performance on DNA material can be categorised according to different criteria. Wandelt et al. [WBL13] suggest to differentiate between algorithms that are suitable to compress a *single*, *continuous* genome sequence and algorithms that are able to compress *multiple separate* raw read sequences which may also require to store quality scores. Further, the mode of operation of the algorithms can be distinguished [LC14]. The class of *horizontal* or *intra-sequence* compressors makes only use of the information contained within the sequence which has to be compressed. In contrast, the class of *vertical* or *inter-sequence* combines at least to sequences to compress a subset of the sequences, e.g. through a referenced based approach.

In addition to the particular mode standard compression techniques can be applied as pre- or post-processing to achieve better compression results. A frequently used pre-processing approach is to apply a *fixed length bit encoding* to the original file in order to store multiple symbols in one byte. Post-processing steps can comprise *run-length encoding* to reduce the number of repetitive symbols, integer compression schemes such as *Golomb coding* or *Fibonacci coding* and *entropy based encodings* like *Huffman coding* and *Arithmetic coding*. Examples for more complex pre- and post-processing steps are the BWT and LZ algorithms which are also commonly used [WBL13].

The rest of this section illustrates the different modes in more detail and introduces existing algorithms that belong to each type. The focus is thereby on recently released or still not superseded algorithms. A more comprehensive list of published compressors is provided by Wandelt et al. [WBL13], Bakr et al. [BS13] and Giancarlo et al. [GRU14]. Moreover, Deorowicz et al. [DG13] present an excellent comparison between algorithms for which the source code has been made available to the public or which are at least available as binary executable. The information about the algorithms in all of these lists is obtained from the original papers which makes them hard to compare since the authors used different test data and targeted different platforms and sources [DG13].

3.1.1 Substitutional

Substitutional algorithms are normally used in horizontal mode. Although it is possible to use them to compress multiple sequences (vertical mode) better results can be achieved by using different algorithms if the sequences have large parts in common. The aim of substitutional algorithms is to compress the data by referencing to earlier detected substrings instead of encoding the repeated sequence twice (Figure 3.1). For this purpose they construct substring dictionaries on-the-fly or use predefined reference strings. Usually the dictionary is implicitly included in the compressed output and has to be

Output	Remaining sequence	Dictionary
—	**G**CGTGATGGC	$A \to 0, C \to 1, G \to 2, T \to 3$
2	**C**GTGATGGC	$GC \to 4$
2, 1	**G**TGATGGC	$CG \to 5$
2, 1, 2	**T**GATGGC	$GT \to 6$
2, 1, 2, 3	**G**ATGGC	$TG \to 7$
2, 1, 2, 3, 2	**A**TGGC	$GA \to 8$
2, 1, 2, 3, 2, 0	**TG**GC	$AT \to 9$
2, 1, 2, 3, 2, 0, 7	**GC**	$TGG \to 10$
2, 1, 2, 3, 2, 0, 7, 4	—	

Figure 3.1 – Lempel-Ziv-Welch compression of a DNA sequence. First the dictionary is initialised with all the symbols of the alphabet. Then in each step the longest prefix which is already in the dictionary is replaced by its code and added to the output. Finally the prefix plus the next character are added as new entry to the dictionary. An implementation of the algorithm is shown in Code B.2.

rebuilt during decompression but does not consume extra storage capacity, e.g. LZ and LZW [ZL77; ZL78; Wel84]. Since this type of compression makes use of integers as indexes for the dictionary a commonly applied post-processing step is to encode these references more efficiently than the pure representation. For instance Golomb coding or Fibonacci coding can be used.

3.1.2 Statistical

Compression algorithms based on statistical models can be considered as an extension to the dictionary-based, substitutional approach [WBL13]. The statistical model is used to encode more frequently occurring substrings with shorter codes, e.g. Huffman coding, or to represent the entire sequence through a single number as done in Arithmetic coding (Figure 3.2). In the former case the "dictionaries" can be implemented for example as probabilistic or prefix trees. In both cases the (initial) model has to be stored in addition to the compressed output. The better the statistical model is able to predict the input sequence the higher compression rates can be achieved. For instance the family of *Prediction by partial matching* (PPM) algorithms makes use of *Markov models* of different orders to predict the best compression for a given symbol [CW84]. The statistical algorithms can be grouped into those that use a static model which is obtained upfront and is not altered during the compression and algorithms which use adaptive models that are updated during compression.

3.1.3 Grammatical

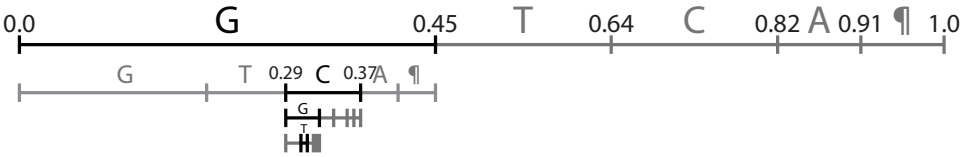
The idea behind grammar-based compression schemes is to represent the input as a *context-free grammar* [KY00]. Since the grammar is able to generate the original input sequence the grammar itself is stored instead of

Lower	Upper	Subsequence
0.0000000000000000	1.00000000000000000	—
0.0000000000000000	0.45454545454545453	**G**
0.2892561983471074	0.37190082644628100	G**C**
0.2892561983471074	0.32682193839218630	GC**G**
0.3063315347312342	0.31316166928488490	GCG**T**
0.3063315347312342	0.30943614134652997	GCGT**G**
0.3088716674164762	0.30915390438150310	GCGTG**A**
0.3089999569460339	0.30905127275785693	GCGTGA**T**
0.3089999569460339	0.30902328231504440	GCGTGAT**G**
0.3089999569460339	0.30901055938649320	GCGTGATG**G**
0.3090067039535989	0.30900863167004605	GCGTGATGG**C**
0.3090084564230963	0.30900863167004605	GCGTGATGGC¶

(a) Compression

Symbol	Probability
A	$1/11 \approx 0.09$
C	$2/11 \approx 0.18$
G	$5/11 \approx 0.45$
T	$2/11 \approx 0.18$
¶	$1/11 \approx 0.09$

(b) Model



(c) Visualisation

Figure 3.2 – Arithmetic coding of a DNA sequence. First a model (b) for the sequence is determined. The model assigns to each symbol in the sequence its frequency. Additionally the symbol "¶" is added to mark the end of the sequence. The initial interval $[0, 1)$ is then divided according to the cumulated frequencies (c). In each step of the compression (a) the subinterval for the particular symbol is chosen and again split up into the regions for the symbols. The complete sequence `GCGTGATGGC` is then represented through any of the numbers in the final interval, e.g. the number 0.309 008 5. An implementation of the algorithm is shown in Code B.3.

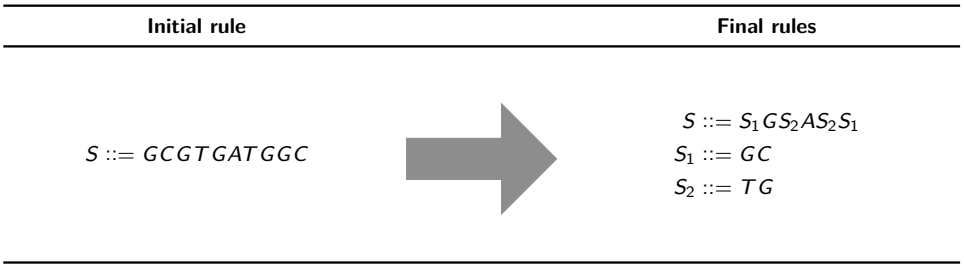


Figure 3.3 – Grammatical representation of a DNA sequence. From the initial rule consisting of the complete sequence subrules are extracted to replace subsequences by shorter codes. The rule extracting is continued in a recursive manner until no more symbols can be grouped to form new, non-trivial rules.

the input (Figure 3.3). To achieve higher compression ratios the grammar can be compressed for example by statistical coders (see Section 3.1.2) [CL04]. Finding an optimal grammar has been proven to be NP-hard [Cha+05] which makes this approach less interesting for compression of DNA sequences. On the other hand an advantage is that this technique supports random access and search in the compressed output [BS13; RK15].

3.1.4 Referential

The referential or relative compression algorithms operate in vertical mode. From a set of sequences at least one is designated as "reference". The other sequences are then encoded by replacing parts common to the references by pointers (similar to using a dictionary) or by just storing INDELs and SNPs (Figure 3.4). The popularity of this compression scheme increases steadily since more and more complete genomes become available and can be used as shared reference. Especially if inter-species references are used high compression ratios can be obtained. For instance is 99.9 % of the DNA is identical between all humans and thus can be replaced by much shorter references. If no suitable reference is available it is also possible to generate artificial sequences and use them as reference for compression [KPZ11]. To achieve high compression rates a widely identical reference and a good mapping onto the reference have to be found. To find the best mapping the reference can be represented as *suffix tree* or *index structure* from which the longest matching parts can be derived. Additional to considering only exact matches there have been approaches to map also reverse substrings, complements or palindromes [MSI00]. Even if this intra-sequence compression technique is used situations can arise where parts of the to be compressed sequence have to be encoded as raw sequence. For instance, this is the case if some symbols do not occur in the reference sequence. Also should the raw encoding be favoured if storing a pointer would result in worse compression, e.g. for very short matches.

	Sequences	Alignment	Output	
Reference	GAGTGTGAC	GAGTG–TGGC	*IN*	5A
Sample	GCGTGATGGC	GCGTGATGGC	*DEL*	—
			SNP	1C, 7G

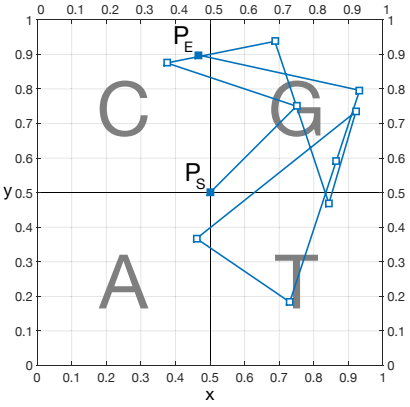
Figure 3.4 – Referential representation of a DNA sequence. First the reference and the sample sequence have to be aligned. From the alignment the INDELs and SNPs can be derived and stored as compressed output.

3.1.5 Chaotic

Already 1990 Jeffrey [Jef90] described in his paper 'Chaos game representation of gene structure' *fractal* structures in DNA sequences. He proposed an unique *Chaos Game Representation* (CGR) for genome sequences which is obtained when the sequence is used as input to the *Chaos Game* (Figure 3.5). Therefore the bases `{A, C, G, T}` span a square and an initial starting point is defined at the centre of the square. For each nucleotide from the input sequence a new point is added into the square. It is positioned exactly half between the last point and the corner corresponding to the nucleotide. In this setting each point represents the subsequence of bases up to this point. Moreover, all points in the same quadrant represent subsequences that end with the base defined by the quadrant. This scheme can by applied in a fractal like style to further subquadrants. He also considers to extend the scheme to encode codons instead of nucleotides and there are recent papers which make use of the CGR to compare DNA sequences [He10; He+12]. Nevertheless, an application of this approach to the area of DNA compression is not often mentioned. A reasonable approach is to only store the last point of the CGR from which the complete sequence can be reconstructed [ANO13]. However, the authors do not provide a functional algorithm using that technique. A problem that they discover is the high precision which is needed to store the floating-point coordinates. Without further post-processing it would be less efficient than the naive 2 bit encoding for each base.

3.2 Objectives of Compression of Genetic Sequencing Data

The information contained in the DNA is used in many different ways. For instance evolutionary relationships can be derived from phylogenetic trees, the understanding of differences between genome sequences improves medical treatments and DNA profiles are used to identify individuals. There does not exists *the best* representation of the DNA which can be used in all these cases. In the same manor there is no "one-size-fits-all" tool to compress the genomic data [GRU14].



x	y	Subsequence
0.50000000000	0.50000000000	—
0.75000000000	0.75000000000	**G**
0.37500000000	0.87500000000	G**C**
0.68750000000	0.93750000000	GC**G**
0.84375000000	0.46875000000	GCG**T**
0.92187500000	0.73437500000	GCGT**G**
0.46093750000	0.36718750000	GCGTG**A**
0.73046875000	0.18359375000	GCGTGA**T**
0.86523437500	0.59179687500	GCGTGAT**G**
0.93261718750	0.79589843750	GCGTGATG**G**
0.46630859375	0.89794921875	GCGTGATGG**C**

Figure 3.5 – Chaos Game Representation of a DNA sequence. The initial point P_S is set to $(0.5, 0.5)$ and for each symbol in the sequence a point is added halfway between the last point and the corner which represents the symbol. Every prefix is then mapped onto an unique point in the graph. The complete sequence `GCGTGATGGC` is encoded in the point $P_E = (0.46630859375, 0.89794921875)$. An implementation of the algorithm is shown in Code B.1.

To structure the field of available approaches Cochrane et al. [CCB12] propose in their paper to find "the best" algorithm for compression according to the importance of the DNA material and its uniqueness. They suggest to store rare or *physical unique* samples of DNA uncompressed and material obtained through experiments which are easy to reproduce and which have low costs with increasing compression factors. For experiments used only to verify existing results they also consider lossy compression and even to store only the outcome of the experiment.

Roguski et al. [RD14b] refer to the "fundamental principles of science" which require experiments to be reproducible and hence do not consider lossy compression as an opportunity. Instead they suggest to differentiate between scenarios which benefit from high compression ratios and scenarios for which a high (de-)compression speed is necessary. When the DNA samples are stored for *long term* archival a high compression ratio and thus a small memory consumption is more important then small (de-)compression times. On the other hand, if the stored data is accessed frequently and used in many experiments the decompression speed has a major impact on the overall performance of the complete experiment.

This scheme can be extended by further scenarios where again different techniques can be more suitable. Instead of increasing the speed of decompression in general it is also possible to aim at only *partial decompression* [BP14] and *random access* [DDG14] to the compressed data. These ideas allow to compress larger collections of sequencing data together to achieve a higher compression ratio. When parts of the compressed material are needed in

an experiment they can be decompressed on their own without the need to decompress the complete collection.

The next step in making the compression of DNA material more useful is to determine possibilities how analyses can be directly performed on the compressed information [GRU14]. This would eliminate the need for decompression and hence reducing the analysis time. Possible applications comprise among others alignment and comparison of compressed sequences and building analysis graphs and self-indexes directly from the compressed material. If more analyses can be extended to operate on compressed data the value of fast *on-the-fly* compression algorithms which can be integrated into the sequencing platforms will increase. Producing the data directly in compressed form makes the storage of raw intermediate data superfluous and can reduce the post-processing time significantly.

4

Algorithms

This chapter describes the details of the algorithms evaluated in this report. First three general purpose compressors are introduced which are used as a baseline for later comparison with the specialised algorithms. The focus for the specialised compressors is on unaligned reads in FASTA or FASTQ format. Compression of already aligned sequence can efficiently be done with a referential scheme and is therefore not topic of this report. In each section one algorithm is introduced and its key features and functionality are shown. All of the mentioned algorithms are available as source code and released under some kind of open source license which allows at least the free usage for evaluation.

4.1 General Purpose Compressors

Although the following three algorithms are not optimised for DNA sequence compression they are widely used [RD14b]. This might be due to unreliable and at best prototypical specialised compressor alternatives. Further, they do not depend on any particular format of input data and achieve at least a high decompression speed. In this report they are used as baseline to compare the specialised compression algorithms against.

4.1.1 gzip

The *gzip* file format [RFC1952] consists of multiple compressed "member" blocks. Each of the blocks stores a part of the compressed input and additionally meta-information and information to verify the consistency of the compressed data. The format itself does not specify how the input is compressed but the implementation of the *gzip* utility – it has the same name as the file format specification – only supports the *DEFLATE* compression.

DEFLATE specification The DEFLATE data format [RFC1951] uses a combination of LZ77 compression and Huffman coding. The input is split up into multiple blocks which are then compressed independently. The LZ77 algorithm uses a sliding 32 kB window to detect repetitions in the input date which can be replaced by pointers consisting of a backwards distance and a length. The resulting stream of distances, literals and lengths is then stored uncompressed or compressed with static or dynamic Huffman codes. To generate the static Huffman codes the *deflate* format specifies to additional rules by which codes of the same length have to be lexicographically ordered according to the corresponding literals and shorter codes must precede longer codes lexicographically. These additional rules make it sufficient to only store the length of the Huffman codes instead of the codes themselves. The dynamic Huffman codes precede the compressed data in the output stream and are again compressed with Huffman coding.

Compression The compressor uses a chained hash table to determine repeated substrings. It compares the hash over the next three bytes of input with the values already stored in the table. If no match is found one byte of the input is copied as literal and the read position is advanced by one byte. In case of a hash match the longest matching substring is determined and encoded as pair of (backwards) distance and repeat length. To favour small distance values the most recently added hashes in the table are checked first. Further, the compressor implements *lazy matching*. Even if a match is found the matcher searches for a longer match starting from the next input byte to find a potentially longer match. At any time the compressor is allowed to remove entries from the hash table or to start a new compression block with cleared Huffman codes.

Decompression The relatively fast decompression is achieved by setting up hierarchical tables to lookup a variable number of bits from the compressed data stream [GA]. This is faster than a normal Huffman tree as more than one bit per level can be decoded. Creating a full table for all codes would consume too much time especially since the *deflate* algorithms produces new Huffman codes very frequently to optimise compression.

4.1.2 bzip2

Data compressed into the *bzip2* format start with a header containing meta-information and an arbitrary number of blocks which are processed independently [Sew07]. The size of the blocks is fixed at compression and can have values between 100 kB and 900 kB. Due to backwards compatibility the first step of the *bzip2* compression is a run-length encoding for each block of data obtained from the input stream. As the author points out it is not needed as the Burrows-Wheeler transform in the second step can

handle pathological cases too. Although the combination of Burrows-Wheeler transform with a Move-to-front transform does not decrease the size of the input directly it is an effective pre-processing step to optimise the overall compression. The actual compression is done by another instance of run-length encoding and Huffman coding. The compression ratio is improved by using multiple Huffman code tables at once and the Huffman code-lengths to reconstruct the tables during decompression are stored via Delta encoding. Further, in intermediate steps values are encoded in an unary format and bitmaps are included to signal which codes are actually used [WikiBzip2].

4.1.3 ZPAQ

The *ZPAQ* file format is designed to support compression of multiple files and enables journaling to create incremental archives of data [Mah14]. The compressed output stream consists of multiple independent blocks and each block comprises multiple segments, e.g. different files. Beside the compressed data stream the blocks and segments contain meta-information about the compressed data. The compression is based on adaptive Arithmetic coding but arbitrary algorithms are allowed as post-processors. To predict the probabilities for the Arithmetic coding various context models are supported which can also be combined into a complete pipeline. The models and the post-processor are initialised for each block whereas the Arithmetic coder is reset for each segment. The algorithms to generate the context and the post-processor are written in a domain specific language *ZPAQL* and directly included in the compressed output. The archive size in journaling mode is further reduced by a deduplication feature.

4.2 fastqz

The *fastqz* program is designed to compress files in the FASTQ format [BM13]. Each read sequence in the file must not exceed a length of 4 095 bases and all lines must have the same length. In addition to the bases `{A, C, G, T}` the symbol `N` is allowed to represent any nucleotide which was not read correctly during sequencing. The information is split up into three streams and compressed through DNA specific encoding (*fast mode*). In the *slow mode* setting the encoded streams are passed to the ZPAQ algorithm (see Section 4.1.3). The predefined context models for the ZPAQ compression algorithm are specific for the compression of sequencing data to optimise the compression ratio of ZPAQ. Each stream is compressed in a separate thread to improve the performance on multi-core machines. A fourth stream is created if an optional reference sequence is specified. This is then used to store only the differences between the reference and the actual sequence. Beside these lossless compression techniques the program allows to quantise the quality values for better compression.

Headers All headers of the read sequences in one FASTQ file begin with an identifier followed by an optional description. Because of this uniformity consecutive headers are referentially encoded in such a way that the latter is described by changes that have to be made to the former header. Digits are stored as incremental offsets and strings are encoded as pair of start and length which has to be copied. Everything which cannot be matched is stored literally. The second header line before the quality values is expected to empty and is therefore not stored.[1] The combined context model used in the *slow mode* comprises two direct and two indirect models which use the column, the byte above and up to four bytes to the left of the current position as context.

Quality Scores The encoding of the quality scores is based on empirically determined frequencies of specific symbols and an inherent structure caused by the sequencing platforms. Although quality values between 0 and 93 are allowed in the FASTQ format the *fastqz* tool restricts the upper limit to 71 to create smaller codes.[2] Depending on their particular values the scores are packed as tuple or triple into one byte, are encoded as single byte. As the sequences tend to start with quality values of 38 a run-length encoding of up to 55 values is used for this score. Trailing scores of 2 are replaced by a single terminating byte. The omitted values can be reconstructed during decoding since the length of the sequence is known (Table 4.1). The context model for the *slow mode* uses three direct models. Their context consists of different parts of the column number and the previous five bytes. They are combined by a weighted mixer using a 14 bit context.

Base Calls The nucleotide alphabet symbols `{A, T, C, G}` are mapped to the numbers from 1 to 4. The symbol `N` for an undetermined base is discarded since it can be reconstructed from a quality score of 0. Depending on the particular subsequence three or four bases can be packed into one byte. A mapping onto the number from 0 to 3 would allow to encode all bases with 2 bit but using different code lengths provides a self synchronisation mechanism. Overlapping reads that start at different offset are likely to be translated into the same byte sequence which allows for better compression (Figure 4.1). When the *slow mode* is enabled three direct and three indirect context models are used. The contexts of the direct models range from 4 to 23 bases and are combined with the indirect models by an order 0 bytewise mixer.

[1] The FASTQ format specifies the second header to be identical to the main header or otherwise empty. Thus the assumption that it has to be empty does not affect the generality of the *fastqz* tool.

Byte encoding	Formulas	Phred scores
0	$\lambda x \,.\, 2$	2 until end of read
1..72	$\lambda x \,.\, x - 1$	$[0, 71]$
73..136	$\lambda x \,.\, ((x - 73) \bmod 8) + 31$ $\lambda x \,.\, \left\lfloor \frac{x-73}{8} \right\rfloor + 31$	$([31, 38], [31, 38])$
137..200	$\lambda x \,.\, (x - 137) \bmod 4 + 35$ $\lambda x \,.\, \left(\left\lfloor \frac{x-137}{4} \right\rfloor \bmod 4\right) + 35$ $\lambda x \,.\, ((x - 137) \bmod 16) + 35$	$([35, 38], [35, 38], [35, 38])$
201..255	$\lambda x \,.\, x - 200$	38 repeated 1 to 55 times

Table 4.1 – Decoding of quality scores in *fastqz* format. Beside the special treatment for the scores 2 and 38 each byte is decoded into one to three quality scores. The score of 38 is handled separately as many sequences tend to start with this value. Trailing values of 2 are replaced by a single zero terminating byte. Since the length of the read is known the number of scores can be reconstructed.

#	Sequence and encoding				
1	TGGA	ATCA	GAT	GGA	ATCA
	11010001	01101101	01000110	01010001	01101101
2	GGA	AATC	AGAT	GGA	ATCA
	01010001	01011011	10000110	01010001	01101101
3		AATC	AGAT	GGA	ATCA
		01011011	10000110	01010001	01101101

Figure 4.1 – Self synchronisation of reads in *fastqz* format. The sequences #1 and #2 are shifted by one nucleotide against each other. But due to the variable encoding of either tree or four bases per byte they synchronise automatically after three bytes. Sequence #3 has an offset of three nucleotides but since they are encoded as single byte the encoded sequence matches the encoding of sequence #2 with an offset of one byte.

Referential Encoding The reference sequence must be provided in a packed format where the nucleotide symbols `{A, C, G, T}` have been replaced by the numbers from 0 to 3. The original sequence must not exceed the size of 4 GB which implies a limit of 1 GB for the packed sequence. The reference is then divided into subsequences with a length of 32 bases from which a hash table is constructed. Alignments are searched independently for each read sequence in the FASTQ file and in both strand directions. The latter increases the chance to find a match with minimal overhead. The alignments are determined by comparing a rolling hash over the read against the reference table. Per match a maximum of four mismatches is allowed and matches are chosen in such a way that long subsequences which few mismatches are favoured. If no match is found or all matches are shorter than half of the read length the entire read is not considered for referential encoding and represented as a single zero byte in the reference stream. Also all bases after the fourth mismatch are encoded literally. In contrast matches are represented as four byte pointer, four mismatch offsets and a direction bit. All matched nucleotides are deleted from the input stream before encoding and optional compression. If the *slow mode* is activated the context used for ZPAQ compression takes the state of the parser, i.e. the mismatch positions and the pointer, into account.

4.3 fqzcomp

The *fqzcomp* tool [BM13] is based on a bytewise Arithmetic coder. The FASTQ input file is split up into three streams which are the headers, quality scores and reads. For each stream a different specifically adapted prediction model is used. Therefore the streams are complete independent of each other and can be compressed in parallel by utilising multiple threads.

Headers Because of the uniform structure the headers are parsed into different tokens from which the model parameters are derived. Tokens are either *literals*, *leading zeros*, *digits* or *separators* including spaces. For each header the immediately preceding header is used as context. If the type of the corresponding tokens matches the values are compared and encoded as *exact match* or as *delta offset* to the previous value for numerical tokens or character for the other types.

Quality Scores The model for compression of quality values takes into account that scores seem to be correlated to previous scores in the read and

[2] Scores above 60 are rarely produced during sequencing (see `http://en.wikipedia.org/wiki/FASTQ_format`, visited on 14/04/2015). Thus the restriction of 71 as maximal value seems not to affect the generality of the *fastqz* tool.

reads as a whole tend to have either high or low quality. In contrast to the FASTQ file format the range of the quality values is restricted to the range of 0 to 62 and a value of 63 is used to signalise a score of 2 repeated until the end of the read. The specific combination of these correlations can be defined through a runtime parameter.

Base Calls The reads itself are compressed with an order-k model such that at each position the previous k bases form the context for the prediction of the current base. The parameter k can be defined at run time. To increase the compression ratio optionally both strands, i.e. also the reverse complement bases, can be used to update the model. The bases `{A, C, G, T}` are encoded as the numbers from 0 to 3. The symbol `N` which represents inaccurate base reads is always encoded as the most frequent code to achieve better compression ratios. During decoding it can be reconstructed since a quality value of 0 is only allowed for this symbol.

4.4 Quip

Quip [Jon+12] is a statistical compression tool which can handle files in FASTQ and SAM/BAM format. It is based on an adaptive Arithmetic coder such that the models are adjusted to the particular data which leads to improvements in compression especially for long sequences. For non-referential compression the input is split up into three streams and four streams otherwise. As the Arithmetic coder strictly separates model and encoding the same coder is used for all streams and only the models are exchanged. Additionally, checksums are calculated and storage for small blocks of the sequences to validate the integrity and the number of reads and bases is stored to provide information without the need to decompress the data.

Headers The header information of the FASTQ files is parsed into tokens of different types and encoded through a mechanism similar to delta encoding. Repeated tokens are replaced by a reference to the token in the previous header. Moreover, non matching numerical tokens are encoded as an offset to the previous number and for literal tokens the longest common prefix with the previous token is determined. All remaining unmatched characters are directly encoded.

Quality Scores The quality scores seem to be highly correlated among each other which makes it sensible to use a Markov model for prediction of forthcoming values. Due to the large alphabet of potential quality scores only an order-3 Markov chain is used to keep the number of parameters small. The context for the Arithmetic coder consists of the last three quality

scores, the position of the current score in the read sequence and the number of adjacent quality values for which the difference was greater than one. These values are binned into fixed intervals to further reduce the number of parameters.

Base Calls As the alphabet of nucleotides is small a Markov chain of order-12 is used to build the model for reads which means that for every position the preceding twelve bases estimate the next base. This model can only develop its full potential if enough training data are available. While it might not be optimal for very short files in general files interesting for compression contain millions of reads and are thus much longer than needed for an adequate training.

Referential *Quip* also supports compression of SAM/BAM formatted files if the same reference as for the alignment is passed as optional argument. Matching read sequences in the SAM/BAM file are replaced by pointers to the reference while unmatched reads are compressed with the same Arithmetic coder when not operating in referential mode.

De Novo Assembly Instead of using an external reference to encode matching reads efficiently *Quip* also supports to create contigs on-the-fly out of a variable number of reads from the actual sequences. This mode of operation can been categorised as a variant of a LZ algorithm as the "code words" are derived from an earlier part of the input and can be reconstructed during decompression. Therefore the assembled contigs do not have to be stored explicitly and the compressed sequence stays entirely self-contained. The de novo assembly is optimised for speed as accuracy only affects the compression rate not the fact of being a lossless compression scheme. Thus k-mers are indexed via an optimised version of a probabilistic Bloom filter (dlCBF) [Bon+06] instead of the widely used De Bruijn graphs[3]. The reads of the input sequence are then aligned to the assembled contigs via a *seed and extend* algorithm and best matches are those with the least Hamming distance.

4.5 Framework For Referential Sequence Compression (FRESCO)

The compression tool *FRESCO* [WL13] uses a referential approach to shrink the size of DNA sequences. In the current implementation only raw sequences over the alphabet `{A, C, G, T, N}` without any meta-information

[3] More information about de novo assembly using De Bruijn graphs is provided in the book *Bioinformatics for Biologists* [PS11].

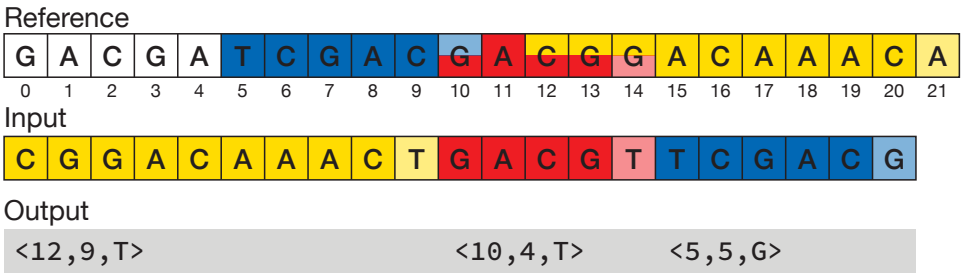


Figure 4.2 – Encoding of reference pointers in *FRESCO*. Matches between the reference and the input sequence are encoded as triples comprising the offset of the matched sequence in the reference, the length of the match and the base which follows the match, i.e. the mismatch after the reference. Although the last match (blue) could also include the last `G` in the sequence it is encoded as mismatch because otherwise a special literal representing a "no mismatch" would be needed.

or formatting are valid input. From a collection of sequences one is chosen as reference and matches between this reference and the other sequences are encoded as a triple containing the offset in the reference, the length of the match and the first mismatching character which occurs after the reference (Figure 4.2). To determine the matches a k-mer hash index is used and in its standard implementation *FRESCO* supports three different strategies to select matches. A greedy search algorithm that uses always the longest possible matches (BAS) and two optimised strategies for fast lookup of local matches (LO) and short local matches in contrast to long matches at a position far away in the reference (LO_MD) respectively. After the decoding a final compression stage features either plain output (PLAIN), i.e. no further compression of the reference triples, delta encoding for the reference triples (DELTA) or a binary compression based on Huffman coding from the *zlib* library (COMPACT).

Reference Selection Finding the best reference for a large collection of sequences is according to the authors a "hard problem" since with a naive approach the number of compressions to find the best reference is quadratic in the number of sequences. In their paper they therefore propose a heuristic to find a good reference at reduced costs. First, one sequence is chosen at random and said to be the base reference against that all other sequences are compressed. Then, the pairwise similarity between all sequences – including the base reference – is determined. The similarity measure is based on the number of match triples which are shared between both compressed sequences. Finally, the sequences are recompressed against the sequence with the highest similarity.

Reference Rewriting If either no good reference is known are simply not existing *FRESCO* is able to create a new artificial reference out of a set of compressed sequences. The new reference is created in such a way that it has a good match covering for the majority of the sequences in the set. To adapt the original reference mismatched SNPs are considered which can easily be derived from the compressed sequence. If a single base change in the reference at a specific position would turn mismatches into matches for a majority of the sequences the rewrite is performed. Most benefit is achieved if the rewrite leads to joining two adjacent reference triples together since the only mismatching base has been changed positively.

Second Order Compression For a collection of sequences not only the sequences themselves are similar but also the mappings onto a reference. Thus *FRESCO* offers an additional post-processing step in which another round of referential compression is applied to already compressed sequences. For each compressed sequence in the collection equal runs of reference triples are searched in a designated reference – also a compressed sequence – and again replaced by a pointer into that reference if a match is found. To handle these large data structures efficiently the comparison is based on hash values similar to a 1-gram-based index.

4.6 Genome Differential Compressor (GDC 2)

The *GDC* compression tool is currently available in two different versions which are not fully compatible to each other. While both target compression of complete sequence collections in FASTA format the earlier version also put a focus on random access to the compressed collection [DG11c]. This has been dropped in the most recent version, namely *GDC 2* [DDN15], to allow for higher compression rates in a second level compression phase. As the authors propose significant higher compression ratios in the later version this report focuses on *GDC 2*.

GDC 2 can be considered as an extension to *FRESCO* (see Section 4.5). Both algorithms share a two-stage compression strategy and use LZ compression[4]. However, *GDC 2* adds various new ideas to push the compression ratio and speed and is compatible to more variations of the FASTA format. The first level of compression uses a hash table with linear probing to perform a LZ factoring between the reference and all sequences. In addition to these *exact matches GDC 2* also searches for *short matches* representing SNPs or INDELs that follow exact matches immediately. As these short matches can be found by just comparing the reference and the sequence without consulting the hash table they improve the compression speed. The encoding

[4] *FRESCO* uses a LZ77 compression whereas *GDC 2* sets on LZSS [SS82] in which the reference is only chosen if it is not longer then a raw encoding.

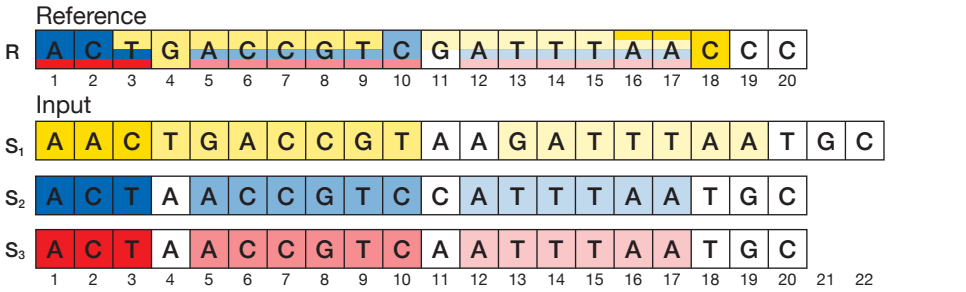
of these matches is chosen such that they do not harm the compression ratio. For a higher compression speed this first stage is completely parallelised by dispatching the compression of each sequence in an individual thread. During the second stage the complete collection of referentially compressed sequences is used as input to another LZ factoring through which common runs of reference tuples are encoded as reference to one of the sequences (Figure 4.3). For this purpose a further lookup table is constructed on the fly which means that only already processed compressed sequences can be used as reference for the second stage.

The encoded values are then further processed by an Arithmetic coder and different type specific methods as for instance delta encoding between actual value and a predicted value for match positions. Collected meta-information about the sequences are stored in an extra file compressed with a standard zlib algorithm. Moreover the reference is encoded with a separate Arithmetic coder.

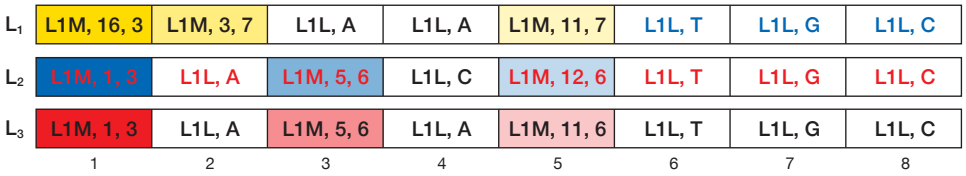
4.7 DNA Sequence Reads Compression (DSRC 2)

The *DSRC 2* framework [DG11b; DG11a] features a standalone application as well as libraries which can be integrated in Python and C++ programs. Primarily it is designed for lossless compression of various kinds of FASTQ format derivatives using statistical compression approaches. But optionally some information can be stored in a lossy fashion. For instance the quality values can be binned into a reduced number of categories as stated in a white paper from the Illumina company [Ill12]. The custom archive format for *DSRC 2* consists of a header containing meta-information about the original FASTQ file and information about the archive itself, a variable number of *blocks* and a footer with information about the different blocks. Each block has an individual header with information about the contained compressed data and about the characteristics which were used during compression. The information of the FASTQ file is divided in three streams and stored in an encoded format following the header of the block. The first version of *DSRC* [RD14b; RD14a] used a slightly different archive format with finer granularity and additional superblocks to allow fast random access. But the authors figured out that this features is not used in many cases and there dropped it in favour of speed improvements. Additionally, in version 2 they added multithreading and support for reading and writing to pipes for a better workflow. Further research on multi-task optimisation for *DSRC* has been conducted by He et al. [He+14].

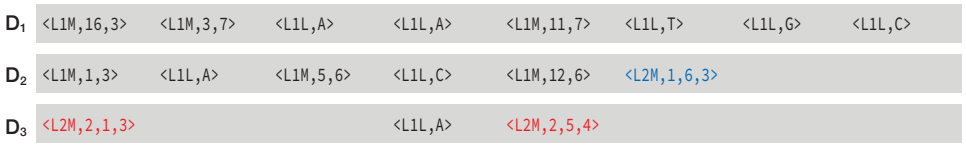
Headers The header information is parsed into tokens with the types *constant*, *numerical* and *non-numerical*. Constant fields are stored only once per block and reference at every occurrence. Numerical fields are delta



(a) Best matches



(b) First level factoring



(c) Second level factoring

Figure 4.3 – Two stage reference encoding in *GDC 2*. Prior to the encoding each sequence is compared to the reference and the best matching subsequences are determined (a). During the first encoding stage (b) the matches are replaced by pointer to a position in the reference. The pointer is specified through a triple consisting of type `L1M`, the offset and the length of the match. All non matched literals are represented through a tuple with type `L1L` and the literal itself. In the second stage (c) each encoded sequence is compared to its predecessors and equal runs of coding tuples and triples respectively are substituted for a second level pointer. The quadruple consist of the type `L2M`, the index of the predecessor in which the run occurred before, the offset and the length of the run. All other codes are copied from the first level encoding.

encoded if it is likely to result in a shorter representation. Otherwise the raw number are stored. For fields containing literal symbols are bit mask is created which signalise which characters in the field do not vary over all instances of this field in the block. These values are only stored once per block. After the individual encodings repetitive values are additionally run-length encoded and an order-0 Huffman encoding is applied.

Quality Scores The compression of quality values depends on the compression mode *DSRC 2* operates in. In the *fast mode* setting either run-length encoding combined with a Huffman entropy coder or an order-1 Huffman coder with the symbol's position as context is used. The used model depends on the particular data and is figured out by the algorithm during runtime. Furthermore, trailing quality values of 2 are stripped to save even more space. In higher compression modes an Arithmetic coder is used. The context comprises the position of the symbol and previous quality value and its size depends on the level of compression.

Base Calls The compressed stream of nucleotides contains only the four base symbols from the alphabet `{A, C, G, T}`. Other symbols which can be used to encode imprecise calls are transferred over to the quality stream. This is only possible if they are marked with low quality scores and otherwise the complete DNA stream of the particular block can only be Huffman encoded. The regular encoding of the base calls depends on the selected mode of compression. In *fast mode* the symbols are just converted in an efficient 2 bit per symbol format. Better compression is achieved if an Arithmetic coder with contextual probability estimation is used. The order of this coder depends on the level of compression that is specified. A formerly – in version 1 – used LZ77 compression is no longer available to achieve higher compression throughput.

4.8 Overlapping Reads Compression With Minimizers (ORCOM)

ORCOM [GDR14] is a disk-based, reference-free DNA sequence compressor. It accepts (potentially gzipped) FASTQ files as input but only processes and stores the base call information from the files. To compress the data it uses a *binning* strategy which groups the reads according to their similarity such that each group can be compressed independently in parallel.

Binning The reads are categorised according to their *minimisers*. A minimiser is defined as the lexicographically smallest p-mer contained in the particular read [Rob+04] and its length is considered to be much smaller than the length of the read. In the *ORCOM* algorithm the chance to find

matches between the reads is increased by finding minimisers also for the reversed-complemented form of the read, i.e. the second strand. To circumvent the problem of an uneven bin distribution due to minimisers the authors derive so called *signatures* from the minimisers which form a restricted subset. After all reads have been processed the bins are lexicographically sorted according to the contained reads which are for the sake of the sorting rotated such that every read has its signature as prefix. This sorting improves the later compression as overlapping reads are stored close to each other.

Compression The compression is performed independently for each bin and thus can be executed in parallel. Per bin a referential compression approach is used which maintains a sliding window over the reads to reduce the amount of potential references. For each read the best reference is determined according to a variant of the Hamming distance with weighted penalties. Once the reference with the smallest distance has been found the read is (conceptually) aligned to the reference such that the signatures overlap. The referential matching data is split into different streams depending on the type of the information. Multiple streams contain meta-information about the reads, the relation between the reference and the read and about the information stored in the other streams. For each symbol of the alphabet `{A, C, G, T, N}` a stream is created that stores the mismatching symbols. The positions of matches are stored in a separate stream and additionally run-length encoded. Further, reads which could not be mapped to a reference sequence are not encoded but the signature is omitted since it is shared between all reads in the bin. The stream containing meta-information and the symbol streams are finally compressed through an order-4 Arithmetic coder[5] while all other streams are handled by the general-purpose compressor *PPMd*[6].

4.9 SeqDB

The *SeqDB* tool [How13] is a FASTQ compressor with focus on high-throughput (de-)compression and backwards compatibility. The former is achieved by using different multithreading techniques and exploiting memory access optimisations. Instead of a pure decompression mode *SeqDB* also features to mount compressed files as FASTQ file through pipes for a simple integration in existing pipelines. The *SeqDB* file format specification abstracts from the concrete storage layout. The only requirement is that the

[5] The Arithmetic coder used is called *Range encoder* which itself is based on a conference paper from 1979 [Mar79]. The source code and further information can be obtained from `http://www.compressconsult.com/rangecoder/` (visited on 18/04/2015).

[6] *PPMd* is based on the *PPMII* algorithm by Shkarin [Shk02]. Further information and the source codes can be obtained from `http://compression.ru/ds/` (visited on 18/04/2015). In *ORCOM* version J rev.1 is used.

storage container must be compatible with two dimensional block access. This requirement allows to provide fast random access into the compressed data stream but also restricts the most efficient use of *SeqDB* to files with fixed read lengths. Nevertheless, variable read lengths can be handled by initially determining the longest sequence and padding all shorter sequences. The authors decided to use the *HDF5* data model [Gro15] in their provided implementation of *SeqDB*.[7] The compression itself comprises two stages: First the data is converted into the internal *SeqPack* format and is then compressed using the *Blosc* (meta-)compressor.

SeqPack For an efficient encoding of base calls and quality scores corresponding items are mapped to an unique position in a 5×51 grid. The bases from the alphabet `{N, A, T, C, G}` specify the x-coordinate in the range from 0 to 4 and the ASCII encoded Phred quality scores form the y-coordinate between 0 and 50. The lookup tables are included in the compressed file such that it is possible to decode the file without having *SeqDB* at hand.

Blosc The Blosc (meta-)compressor[8] is essentially a framework to optimise the data flow for compression tasks. It uses efficient cache-blocking techniques and multithreading to provide a high-throughput compression pipeline. In its standalone version it also ships with a shuffle pre-processor using SIMD instructions and the *BloscLZ* compression algorithm. This is based on a LZ77 variant called *FastLZ*[9] and is also highly optimised for compression speed.

4.10 Sequence Compression Algorithm Using Locally Consistent Encoding (SCALCE)

Instead of providing a complete compressor the aim of *SCALCE* [Hac+12] is to *boost* the performance of existing algorithms in compressing read sequences. In their implementation the authors support bzip2 and (parallel) gzip as back end. To achieve better compression *SCALCE* clusters the reads of the FASTQ file according to their similarity. Rather than to compare the complete reads *SCALCE* uses a variant of *Locally Consistent Parsing* (LCP) [SV96] which is optimised for speed. For each read *core substrings* are determined by finding core blocks for distinctive *marker symbols* and projecting the concatenation of these blocks into the substring space. The search for all cores is implemented

[7] Independent of the work on *SeqDB* Mason et al. [Mas+10] proposed a new standardised NGS file format *BioHDF* which is an extension to the pure HDF5 storage container.
[8] Further information about Blosc is provided at `http://www.blosc.org/` (visited on 18/04/2015).
[9] For further information about FastLZ see `http://fastlz.org/` (visited on 18/04/2015).

via the *Aho-Corasick* matching algorithm [AC75] such that substrings are found in parallel. The cluster is determined by the longest substring and if two substrings of the same length are found the already bigger cluster is preferred.

Since the quality scores are not the primary target of *SCALCE* they are simply compressed via an order-3 Arithmetic coder. Optionally a lossy reduction of the quality values onto their local maxima is provided which according to the authors improves the compression without reducing the usability of the data. For the same reason read identifiers are just copied into a different file and compressed via one of the back end compressors.

4.11 kpath

The *kpath* [KP15] compressor is based on an adaptive Arithmetic coder to compress the reads of a FASTQ file – all other information than the read sequences is discarded. As bootstrapping for the Arithmetic coder an initial model is obtained from a provided "gzipped" FASTA reference. In contrast to other referential algorithms the reference used by *kpath* does not have to cover the entire to be compressed reads in the input file as it is not used for alignment. Even if a non-optimal reference is selected the algorithm will by good chance benefit from the reference and will converge to the actual data during the compression. During a preprocessing stage reads can be converted to their reverse complement if this is better represented through the reference, the reads are reordered to group similar sequences together in the file and exact duplicates are stored only once. From each read an initial k-mer (*Head*) is split off and encoded through a graph-based approach while the rests (*Tails*) are fed into the Arithmetic coder.

Read Heads The heads of length k are represented as a traversal in a complete 4-ary tree of depth k, i.e. a graph which contains every possible k-mer exactly once as path from the root to a leaf. The traversal is encoded as a single bitstring according to two rules. Every time a node is visited and is contained in at least one of the heads a 1 is added to the bitstring. If the node would create a path that does not represent a head a 0 is added instead. Finally the bitstring is compressed with the gzip algorithm. To restore the correct number of heads an additional file is created containing the counts for each head. Otherwise duplicates would be lost due to the single representation in the tree.

Read Tails The initial probability distribution for the Arithmetic coder is obtained from the reference. First a de Bruijn graph is created out of all the reads in the reference. In this graph each k-mer is represented through a node an edges are drawn between nodes that overlap in $k - 1$ nucleotides.

Then the (cumulative) probability distribution for all edges is estimated from this graphs. It specifies which sequences are used more commonly than the others. The update of the model can be seen as an order-k Markov chain for which the probabilities are updated as the graph is traversed during the compression phase. Moreover, the model is constructed only over the bases `{A, C, G, T}`. Every occurrence of the additional symbol `N` is treated as `A`. The correct reconstruction could be achieved by replacing every `A` with an `N` if the corresponding quality score has the lowest possible value.

5

Evaluation

The evaluation covers all compressors introduced in Chapter 4 over a collection of different datasets. The first part of this chapter outlines the environment and the data which have been used for the evaluation. The second part lists and describes the obtained results.

5.1 Hardware, Software and Test Data

The focus of this report is to evaluate the usability of the introduced tools in an environment which not exclusive to large institutions with high-performance architectures. Since the sequencing platforms themselves get affordable for an increasing number of institutions the tools to process the produced should not harm this positive technological change. Instead of the high-performance servers with large memories used in most articles the tests have been run on desktop computers featuring an Intel Haswell processor and 16 GB of main memory. Detailed information is shown in Table 5.1.

On the computers *Ubuntu 14.04.1 LTS* has been installed as operating system and all programs have been compiled with the *GCC (Ubuntu 4.8.2-19ubuntu1)*. In addition to possible program specific options the flag `-O3` for heavy optimisations and *SSE* and *AVX* support have been enabled during compilation. Aside from *FRESCO* which is the only algorithms that only supports multithreading via the `Boost.Thread` library all tools with parallel execution have been compiled with native C++11 thread support. The *FRESCO* tool has been adapted to accept a list of files as in put for the first compression stage instead of an input directory, unnecessary copying of the reference sequence has been prevented and the removal and recreation of the output directory has been disabled. The *GDC 2* algorithm was originally only able to compress FASTA files which had the same number of reads. By explicitly storing the number of reads of each file in the already

Processor Number	i7-4770
Last Level Cache	8 MB
Cores	4
Threads	8
Processor Base Frequency	3.4 GHz
Max Turbo Frequency	3.9 GHz
Instruction Set Extensions	SSE4.1/4.2, AVX 2.0
Main Memory Size	16 GB
HDD Throughput	~166 MB/s

Table 5.1 – Processor and memory statistics for the evaluation system. The facts about the 4th generation Intel® Core™ i7-4770 processors are taken from `http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz`. The hard disk drive throughput has been estimated via the command `dd if=/dev/zero of=./output conv=fdatasync bs=512k count=1k`. Synchronisation is needed to eliminate side effects due to caching.

existing description file increases the generality of the tool with a minimal overhead. At the moment the number of reads for each file is stored as four byte unsigned integer. However, this changes the archive format and is not backwards compatible to older versions. In Table 5.2 the versions of the specialised DNA compressors used are listed and Table A.3 provides additional information about the locations of the source codes.

The test data have been chosen across different species to measure the performance on various DNA structures. However, for each species always more than one sample is available to explore the benefits of referential compression of collections of similar sequences. Furthermore the data comprises different sequencing techniques such that an influence on the compression can be evaluated. All sequencing data have been obtained in FASTQ format as this contains all the information needed. Since *fastqz* and *fqzcomp* enforce the quality score for the symbol `N` to be zero all files have been pre-processed to replace non-zero scores. The altered files have then been used for all evaluations. For *GDC 2* these files have been converted to FASTA format by omitting the second ID line and the quality scores. Further, for *FRESCO* all meta-information has been stripped as it can only handle raw sequences as input. A short overview over the data is provided by Table 5.3 and a detailed listing of all sequences can be found in Table A.2.

Compressor	Version	Last update
fastqz	1.5	15/03/2012
fqzcomp	4.6	23/05/2013
Quip	1.1.8-4-g337ca2a	31/03/2015
FRESCO	9ae6dbcb24[a]	04/12/2014[b]
GDC 2	2.0	05/03/2015[b]
DSRC 2	2.0 RC2[c]	10/12/2014
SeqDB	0.2.1	05/01/2015
ORCOM	1.0rc	01/12/2014
SCALCE	1	21/04/2015
kpath	0.6.3 (1-6-15)	06/01/2015

[a] ID of the last commit as no official version number is available.
[b] Date of the last official update. Sources have been changed within the scope of this report.
[c] The branch *dna-extra-symbols-fix* has been used.

Table 5.2 – Version information of the evaluated compressors. Unless otherwise stated the sources have not been altered and compiled as specified in the corresponding documentation.

Species	Num. of sequences	Avg. size on disk [MB]
Arabidopsis thaliana	5	243
Escherichia coli	10	838
Gorilla	2	30 363
Homo sapiens	5	13 954
Mus musculus	5	2 731
Ceratotherium simum	3	39 645

Table 5.3 – Summary of the species represented in the evaluation. The data have been chosen to represent various DNA structures and different sizes to measure the generality of the algorithms. The average file size relates to the pre-processed FASTQ files.

	gdc2	orcom	fresco	seqdb	dsrc-m0	dsrc-m2	quip	quip-a	fqzcomp	fastqz	fastqz-r	scalce	kpath
A. thaliana	✓	✓	✓[a]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
E. coli	✗[b,c]	✓	✗[c]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Gorilla	✗[d]	✓	✗[e]	✓	✓	✓	✓	✓	✓	✓	✗[g]	✓	✗[b]
H. sapiens	✗[d]	✗[?]	✗[e]	✓	✓	✓	✓	✓	✓	✗[f]	✗[f]	✗[f]	✗[b]
M. musculus	✗[d]	✓	✗[e]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗[b]
C. simum	✗[d]	✓	✗[e]	✓	✓	✓	✓	✓	✓	✓	✗[g]	✓	✗[b]

[a] After the limit of 12 hours only the first compression stage was finished.
[b] Not enough RAM (not crashing).
[c] Manually aborted after 12 hours.
[d] Not enough RAM (`std::bad_alloc`).
[e] "Killed" by the operating system due to excessive memory consumption.
[f] Variable number of base calls not supported.
[g] Reference larger than 1 GB not supported.
[?] Unknown error during execution.

Table 5.4 – Summary of algorithm failures. The checkmark (✓) indicates that the corresponding algorithm completed without any error on the dataset. An x mark (✗) represents an error during runtime. The reason for the abort is specified in the table notes.

5.2 Results

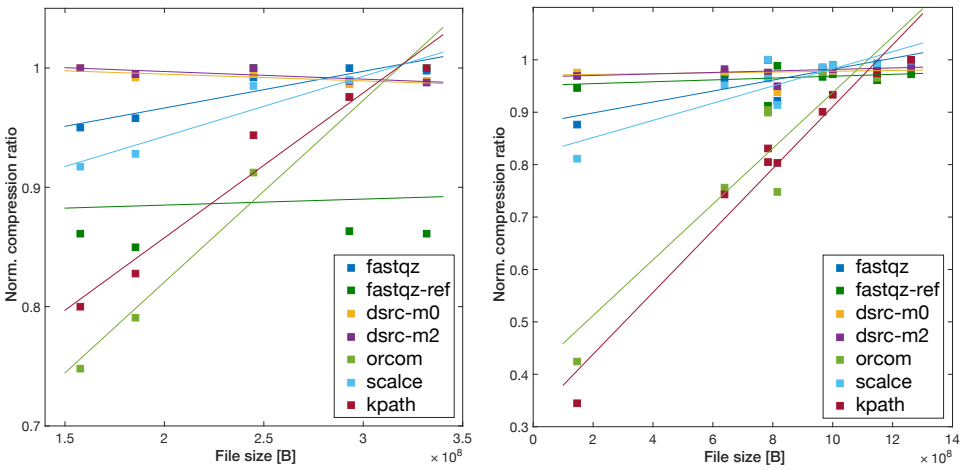
This section provides an overview over the results produced within the scope of this report. Each finding is briefly described and backed up by measurements and statistics.

Problems during execution Due to the different requirements that the compressors impose on the data and on the hardware not all algorithms succeeded on all datasets. Some problems occurred because of structural problems with the input files. For example did not all compressors allow the read length to differ throughout the file. Nevertheless was the majority of errors caused by insufficiently careful memory management. Table 5.4 summarises the various problems which caused the algorithms to fail.

Influence of the file sizes With respect to the average over all datasets and algorithms the results do not show a significant correlation between the size of the data files and the compression ratio (see Table 5.5). In contrast, the evaluation showed that depending on the algorithm the compression ratio indeed correlates with the size of the file (see Figure 5.1). For instance for the *ORCOM* algorithm smaller files tend to have smaller compression ratios whereas the *DSRC 2* compressors are not affected by the size.

Dataset	Avg. file size [GB]	Avg. Norm. compression ratio
A. thaliana	0.24	0.74
E. coli	0.84	0.78
Gorilla	30.36	0.81
H. sapiens	13.95	0.77
M. musculus	2.73	0.81
C. simum	39.65	0.76

Table 5.5 – Normalised compression ratio across the datasets. The compression ratio seems not to be correlated with the size of the files in the datasets. For each algorithm the compression ratio has been normalised over all datasets. The average is performed over all algorithms and all files in the particular dataset.



(a) A. thaliana **(b)** E. coli

Figure 5.1 – Correlation of file size and compression ratio per dataset. For both datasets *A. thaliana* (a) and *E. coli* (b) the algorithms *ORCOM* and *kpath* show a strong linear positive correlation between the file size and the compression ratio. The parameters are still correlated for the compressors *fastqz* and *SCALCE* whereas the figures show no significant correlation for *DSRC 2* and the referential approach of *fastqz*.

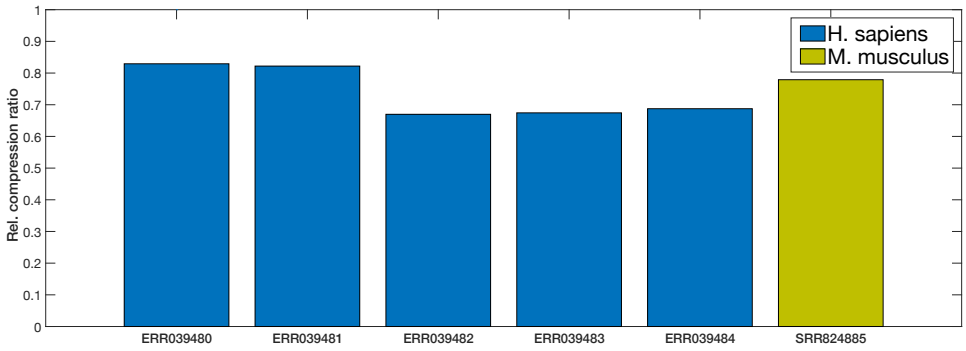


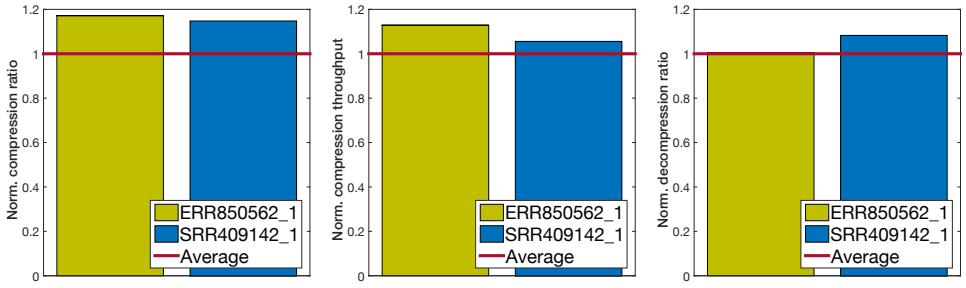
Figure 5.2 – Influence of variable read lengths on the compression ratio. Files containing reads with a variable number of base calls tend to have smaller compression ratios than for files with reads of fixed length. The visualised compression ratios correspond to files with variable read lengths. The compression ratios are relative to a normalised average compression ratio of 1 obtained from the files with reads of fixed length.

Influence of variable read lengths The comparison in Figure 5.2 shows worse than average compression results for files that contained reads of variable lengths. The biggest fall-off in the compression ratio of variable length reads can be noticed for three files in the *H. sapiens* dataset which comprises read lengths between 4 and 2 876. Their relative compression ratio is around 40 % below the average compression ratio of a control group of files with fixed read lengths for the human genome.[10] The compression ratio of the only file in the *M. musculus* dataset with variable read lengths is only about 78 % as high as the compression ratio of the other files in this dataset.

Outliers among the datasets Despite the deviations due to the file size and the read length the files *ERR850562_1* (*M. musculus*) and *SRR409142_1* (*C. simum*) exhibit significant better than average compression ratios and speeds (see Figure 5.3). The decompression throughput for the file *SRR409142_1* is only similar to the average. This is due to the significant worse results of *drsc-m0* and *seqdb* on this file (on average both obtain at most 29 % of the average performance, data not shown).

Compression throughput The compression throughput for all algorithms on each dataset is shown in Table 5.6. The highest throughput of 537 MB/s is achieved by *SeqDB* on the *E. coli* dataset. Over all datasets *SeqDB* and

[10] The dataset of the species *H. sapiens* does not contain files with reads of fixed length. Thus for the purpose of this statistic an additional group of files with similar characteristics has been evaluated. The identifiers of the files are ERR018416_1, ERR018417_1, ERR018418_1, ERR018419_1, ERR018420_1, ERR018421_1 and ERR018422_1.



(a) Compression ratio **(b)** Comp. throughput **(c)** Decomp. throughput

Figure 5.3 – Outliers among the datasets. The files *ERR850562_1* (*M. musculus*) and *SRR409142_1* (*C. simum*) have significant better compression characteristics than the average. While the compression ratio (a) and the compression throughput (b) of both files are above the average the decompression throughput (c) of the first file is slightly below the average. All sizes have been normalised according to the respective dataset.

DSRC 2 have the highest performance whereas *GDC 2* and *FRESCO* are significantly slower than all other algorithms. Among the general purpose compressors *ZPAQ* is approximately five times faster than *gzip* and *bzip2* which puts it in the middle of the field. The global average of the compression throughput evaluates to 79 MB/s.

Decompression throughput Table 5.7 provides an overview over the decompression throughputs for all algorithms on the different datasets. The maximum throughput of 588 MB/s is reached by *SeqDB* on the *A. thaliana* dataset but a similar high throughput is reached by *ORCOM* on the *E. coli* dataset. Also on the other datasets *ORCOM* shows high throughputs between 200 MB/s and 500 MB/s and shares the first place with *DSRC 2* and *SeqDB*. The compressors *fastqz* and *kpath* are placed on the low ranks with a throughput around 6 MB/s to 9 MB/s. Out of the group of general purpose compressors *gzip* features the shortest decompression times and its throughput of 142 MB/s is significantly above the global average of 122 MB/s.

Compression ratio The compression ratios cannot be compared across the different types of compressors as for instance quality values tend to be harder to compress than nucleotide sequences. Therefore the compressors in Table 5.8 are grouped according to the type of their compressed output. In the group of the fully featured FASTQ compressors the referential version of *fastqz* achieves the highest compression ratio of 6.77 on the *A. thaliana* dataset. For the other species also *fqzcomp* and the non-referential version of *fastqz* obtain compression ratios above 5. Remarkable is the low ratio of *SeqDB* which lies with 2.27 even below the results of the general purpose

	A. thaliana	*E. coli*	*Gorilla*	*H. sapiens*	*M. musculus*	*C. simum*
gzip	16.25	14.65	15.87	14.18	17.93	14.51
bzip2	16.25	15.45	14.85	16.33	13.78	14.88
zpaq	84.09	93.35	76.60	73.36	70.70	72.09
fastqz	10.60	10.86	9.14	—	—	8.54
fastqz-r	8.72	6.84	—	—	—	—
fqzcomp	85.93	60.91	59.56	60.15	63.26	54.03
quip	65.84	68.72	53.64	49.06	51.71	48.48
quip-a	18.71	11.08	22.66	23.13	29.11	32.48
dsrc-m0	395.38	426.24	*142.53*	*135.51*	166.05	*139.47*
dsrc-m2	34.14	72.25	128.44	126.18	80.31	128.08
seqdb	*464.88*	*536.73*	100.50	79.66	*439.29*	115.67
scalce	32.10	46.82	33.49	—	—	21.81
gdc2	0.01	—	—	—	—	—
orcom	56.73	124.91	57.60	—	—	63.97
kpath	7.72	9.32	—	—	—	—
fresco	0.02	—	—	—	—	—

Table 5.6 – Compression throughput overview. The compression throughput is measured in MB/s and defined by input size divided by runtime. The throughputs represent the average over all files for each dataset. The highest throughput for each dataset is emphasised.

	A. thaliana	*E. coli*	*Gorilla*	*H. sapiens*	*M. musculus*	*C. simum*
gzip	181.50	180.20	88.64	127.05	184.27	93.11
bzip2	37.83	35.57	31.48	36.06	39.21	32.28
zpaq	111.03	116.93	95.85	82.53	108.60	66.18
fastqz	9.56	9.70	8.75	—	—	8.57
fastqz-r	9.51	9.71	—	—	—	—
fqzcomp	68.17	69.81	53.27	53.59	77.72	50.71
quip	35.08	34.83	31.11	27.18	40.18	27.51
quip-a	15.04	9.94	30.48	23.30	28.50	27.46
dsrc-m0	503.05	497.26	127.88	*137.28*	*404.58*	127.97
dsrc-m2	29.59	71.61	118.39	109.00	79.07	103.78
seqdb	*588.09*	536.38	92.61	68.84	354.14	89.97
scalce	64.82	75.76	59.41	—	—	57.44
gdc2	—	—	—	—	—	—
orcom	338.13	*576.40*	*200.75*	—	392.70	*214.79*
kpath	5.69	6.50	—	—	—	—
fresco	21.05	—	—	—	—	—

Table 5.7 – Decompression throughput overview. The decompression throughput is measured in MB/s and defined by output size divided by runtime. The throughputs represent the average over all files for each dataset. The highest throughput for each dataset is emphasised.

	A. thaliana	*E. coli*	*Gorilla*	*H. sapiens*	*M. musculus*	*C. simum*
gzip	2.98	2.82	3.16	3.01	3.20	2.90
bzip2	3.77	3.45	4.01	3.86	4.02	3.63
zpaq	2.71	2.51	2.84	2.78	2.88	2.61
fastqz	5.82	5.88	*5.56*	—	5.70	*5.39*
fastqz-r	*6.77*	*6.25*	—	—	*5.94*	—
fqzcomp	5.04	4.61	5.38	4.73	5.37	4.78
quip	4.93	4.61	5.35	4.35	5.14	4.86
quip-a	4.93	4.85	5.43	4.35	5.14	4.90
dsrc-m0	4.13	3.88	4.45	3.92	4.47	4.07
dsrc-m2	4.70	4.30	5.13	*4.66*	5.01	4.69
seqdb	2.26	2.25	2.32	2.17	2.46	2.18
scalce	5.32	5.68	5.40	—	5.33	5.29
gdc2	2.68	—	—	—	—	—
orcom	*13.23*	*32.10*	*14.44*	—	*13.35*	*18.45*
kpath	12.49	30.54	—	—	—	—
fresco	*7.72*	—	—	—	—	—

Table 5.8 – Compression ratio overview. The compression ratio is defined as the uncompressed file size divided by the compressed size. The uncompressed file size is chosen according to the type of the compressed output, e.g. *fresco* only compresses raw sequences and thus the size of the uncompressed raw sequence file is used. The ratios represent the average over all files for each dataset. The algorithms are grouped according to the type (FASTQ, FASTA, raw sequence) of the compressed output and the highest ratio for each group per dataset is emphasised.

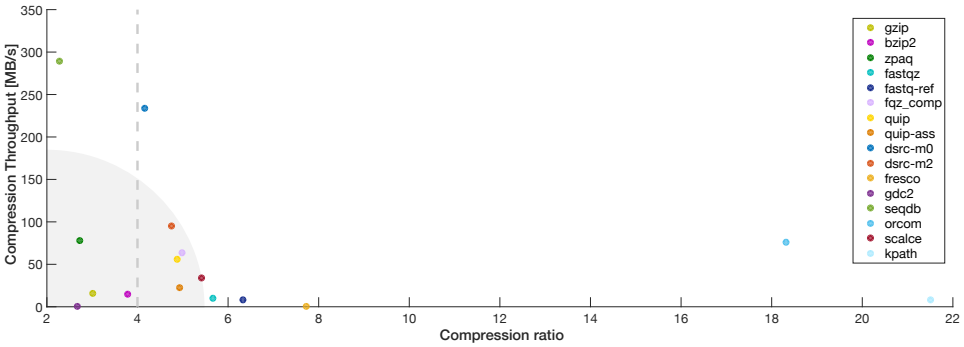
compressors. Among the FASTA compressors *ORCOM* and *kpath* share the highest results of ratios between 10 and 30. The only successful evaluation of *GDC 2* on the *A. thaliana* dataset yields a significantly worse ratio of 2.68. *FRESCO* is the only algorithm which can only compress raw sequence information but has with 7.72 also a lower compression ratio than the FASTA compressors.

6

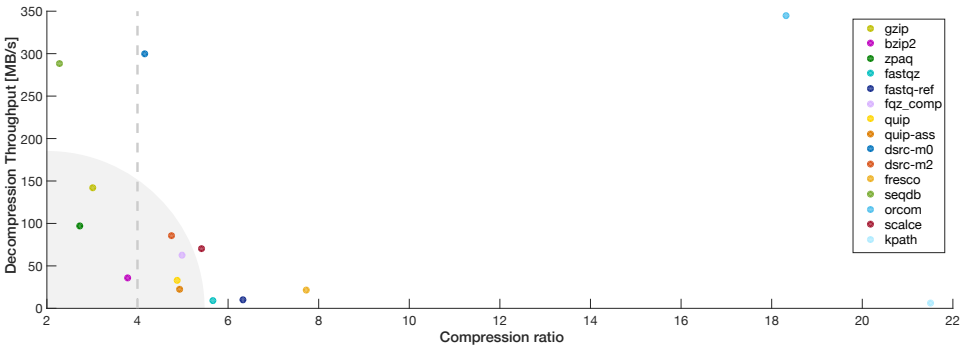
Discussion

During the evaluation of the different compressors it became clear that only very few of them can be considered as a stable, well-documented and ready to use tools. For instance *fastqz* and *fqzcomp* were created as proof of concept within the scope of the Sequence Squeeze competition [HL13]. While providing new insights and contributing ideas in the field of genetic sequencing data compression their development seems to be discontinued (see Table 5.2). Moreover, on large datasets *ORCOM*, *FRESCO* and *GDC 2* crashed, got "killed" or exited in a controlled way (see Table 5.4) but without any helpful information about the cause. Despite from incompatibilities during compilation of *DSRC 2* this algorithm as well as *Quip* and *SeqDB* made a good impression and especially the latter ones provide a reasonable documentation.

Most of the tools were able to operate reasonably well with the limited amount of 16 GB of available RAM. Only *GDC 2*, *FRESCO* and *kpath* failed to compress all but the smallest files. While *kpath* simply slowed down as the system was running out of free memory *GDC 2* and *FRESCO* either tried to allocate more memory than the remaining free space and crashed or got "killed" by the system due to excessive memory usage. The authors of *kpath* already warn in their paper that much memory is needed to construct the required data structures and also *GDC 2* has always been executed with more available memory. But especially for *FRESCO* the failure surprised as the authors claim in their paper to have run all their experiments on a laptop with no more than 2 GB RAM. Aside from the *A. thaliana* dataset for which the first stage completed after 1.3 hours all other runs have been manually aborted after 12 hours without any result. Among the runs which finished on this dataset only *GDC 2* was slower with 2.6 hours.



(a) Compression performance



(b) Decompression performance

Figure 6.1 – Overall performance of the compressors. In (a) the compression ratio is plotted against the compression throughput and in (b) it is plotted against the decompression throughput. An optimal algorithm would be positioned in the right upper corner achieving a high compression ratio with little runtime. The dashed grey line indicates the compression ratio which can be achieved by converting raw sequences in ASCII format into a 2-bit-representation. The light grey areas visualise Pareto fronts for better comparison of the algorithms.

The next two sections discuss the results regarding the compression ratio and the (de-)compression throughput respectively. A comparison of the overall performance of all algorithms is illustrated in Figure 6.1.

6.1 Compression Ratio

The obtained results cover two different areas which have to be discussed. Primarily, the different compression techniques have to be compared against their performance. But the results also show an influence of the files which are going to be compressed.

6.1.1 Compression Techniques

Comparing the full FASTQ compressors against those which only aim to compress FASTA files or raw sequences would not be fair. Due to the larger alphabet size the quality values in FASTQ files tend to be harder to compress [Hac+12].

The overall highest compression ratio among the FASTQ compressors has been obtained with the referential compression mode of *fastqz*. The compressors was able to shrink the FASTQ file to nearly one seventh of its original size. Moreover, on all datasets on which the referential mode of *fastqz* has been evaluated it placed first outperforming all other FASTQ compressors in terms of the compression ratio (see Table 5.8). The better compression ratio is founded in the fact that instead of storing the sequences themselves only pointers to positions in the reference are stored. If long matches are found during compression storing only the "address" of a match significantly outweighs storing the complete matching sequence. Furthermore, the model used for compression of quality scores must be efficient in *fastqz* as these values contribute to a large portion of the compressed files [Jon+12]. This assumption is supported by fact that *fastqz* in non-referential mode is ranked just below the referential mode in terms of the compression ratio.

Despite the naturally higher compression ratios that referential compression algorithms achieve they come along with the risk of loosing all information about the compressed content if the *external* reference gets inaccessible [Jon+12]. A promising alternative is to create a self-contained archive from a collection of files instead of compressing single files on their own. A subset of the collection is then used as meta-references to compress the remaining sequences. These meta-references can later be compressed by a different, independent algorithm. This scheme is implemented by the FASTA compressors *GDC 2* and *FRESCO*. While *FRESCO* obtains at least a compression ratio higher than the baseline of two bit per base call *GDC 2* was only able to reduce the file size by a factor of 2.7. These results contrast with the ratios reported in the respective papers about the algorithms. Possible reasons for the differences might be the smaller amount of RAM that was available and the small size of the datasets. It can be expected that the performance of this kind of algorithms increases with the size of the collection as more reference material is available.

The approach of using these kind of meta-references is nevertheless promising. The tool *ORCOM* achieves high compression ratios by using single reads within one file as reference instead of searching for them throughout a collection of files. This is further supported by grouping reads according to their similarity which is implemented in *ORCOM*, *kpath* and *SCALCE*. The benefits of this strategy are twofold. On the one hand it directly increases the compression ratio as the offset to reference positions can be encoded through shorter numbers. On the other hand it helps to maintain the same

compression ratio while using smaller blocks or window sizes since good reference are closer to the matching sequences are not overseen due to small windows. In the evaluation both *ORCOM* and *kpath* achieved much higher compression ratios than *FRESCO* and *GDC 2* but this result has to be used with caution since the latter ones seem not to be fully functional. The FASTQ compressor *SCALCE* placed after the *fastqz* compressors by just using this boosting technique and basic Arithmetic encoding while *fastqz* concentrates on effective statistical models.

Instead of using an existing reference *Quip* performs a de novo assembly with the first reads of the be compressed files. The obtained contigs are then used to referentially encode the remaining sequences. Similar to the previous approaches this creates a self-contained archive while also profiting from the benefits of a referential compression. The compression ratio of 4.9 is slightly higher than the ratio in the normal mode and puts *Quip* in the middle of the field.

As expected the general purpose compressors occupy the last places with respect to the compression ratio. Since *SCALCE* first applies its boosting strategy and then uses *gzip* compression the worse performance of *gzip* alone affirms the effectiveness of forming clusters within the reads. However, it might be worth to apply the boosting to other per se better back end compressors to get even higher compression results. Surprisingly *SeqDB* has a compression ratio worse than the general purpose compressors. But this might be due to the explicit focus on speed rather than on compression.

6.1.2 File Structure

The strong positive correlation between file size and compression ratio for the algorithms *ORCOM* and *kpath* (see Figure 5.1) can be explained by the clustering and compression strategies that both algorithm use. The larger the files are the higher are the chance to find similar reads and the larger the clusters get. These two factors directly influence the referential compression technique of *ORCOM* and the Arithmetic coder used in *kpath* which achieve better compression results for larger clusters. For *SCALCE* the correlation is not that high because the used *gzip* back end compressor can only benefit indirectly from the better clustering. The compression improves only if the number of similar reads that fit into the sliding window of the *gzip* algorithm increases. Since *DSRC 2* does not perform any reordering of reads it does not profit from larger files.

Further, the results indicate worse compression ratios for files containing reads with different lengths. On the dataset chosen in this report the effect was visible throughout all evaluated algorithms (which are able to handle variable read lengths). On the dataset for the species *M. musculus* a direct comparison was possible as it contained file with uniform reads as well as well variable read length. For the species *H. sapiens* a separate

dataset with uniform reads has been used to validate the result. However, to obtain founded results the effect should be studied on larger datasets and across more different species to rule out other potential causes such as the distribution of the read lengths or the length of the reads itself. At the moment one plausible explanation for the results is a worse behaviour of the used statistical algorithms on reads with variable lengths. It might be the case that the prediction of the derived models is better suited for reads of the same length. Moreover there might be a direct connection between the worse compression ratio and the biological reason that caused the shorter reads during sequencing.

6.2 (De-)Compression Throughput

In general a high compression ratio implies low (de-)compression throughput. This seems to be obvious as lower compression ratios imply further that less work has to be done to achieve the ratio. Moreover, on average the decompression throughput is higher than the compression throughput. While for compression the files had to be parsed and models had to be obtained these pre-processing is not necessary for decompression which reduces the work that has to be done. Especially for *ORCOM* this has a major impact. During compression the binning algorithm has to be run which accounts for approximately one third to one half of the compression time and the best reference for each read has to be determined. As these steps are not necessary for the decompression its throughput is more than three times higher.

Except for the compression throughput of *SeqDB* the multithreaded tools obtain the highest speeds. While *SeqDB* has no practical use in terms of compression due to its small compression ratio both variants of *DSRC 2* are during compression at least on par with slower but more efficient compressors, e.g. *SCALCE*. With respect to decompression *SCALCE* benefits from the high decompression throughput of *gzip* (an LZ algorithm) and thus ranks now better compared to *DSRC 2*.

The referential compressors *fastqz*, *kpath* and the de novo assembly mode of *Quip* have a comparably low throughput which is attributable to the search for matches between the reference and the to be compressed sequences or the assembly step respectively and more complex compressors back ends. For instance is Arithmetic coding known to be quite slow in comparison to other statistical compression techniques.

6.3 Applications in Hardware

As described earlier with the advent of NGS techniques the rate at which new experimental results are produced has overtaken Moore's law. While

currently the amount of data can be handled by distributing the workload of for instance data compression over multiple cores or processors this approach must finally reach a point where simply no more parallelism is possible or at which the energy consumption of such system is no longer sustainable. The idea is to replace the general purpose processors with specialised *Dataflow engines* like *FPGAs*.

Because of the limited resources that are available on these devices the focus has been on implementations of substitutional compression algorithms such as LZ variants. They operate by design in blocks or use a sliding window which creates an upper bound for the needed working memory and only require a single pass over the input data which allows for efficient pipelining and a high throughput. Li et al. [Li+14] and Leavline et al. [LAS13] propose a hardware implementation of the *LZMA* algorithm which is a more sophisticated variant of the *LZ77* algorithm and makes use of a range coder as post-processor. A parallel implementation of *LZW* is provided by Cui et al. [CW08]. The parallel design is enabled by using a hierarchy of small dictionaries instead of a large global one. This also circumvents the problem of slow, large string comparisons for later added dictionary entries. In their paper the author claim to reach a peak throughput of approximately 200 MB/s which would be competitive to the evaluated software based compression algorithms. Finally a fully featured *gzip* implementation is presented by Rigler et al. [RBK07] and Rigler [Rig07] but the authors remark that their design has to be further improved to be competitive with software based implementations.

While these general purpose algorithm on their own have not been competitive in compressing genomic data (see results of *gzip*) they achieved good results when combined with boosting techniques prior to the compression (see results of *SCALCE*). In *SCALCE* the boosting consists of finding core substrings for the reads and a pattern matching based on the Aho-Corasick algorithm. As determining the core substrings is a simple linear operation on the reads and Vidanagamachchi et al. [Vid+11] and Chen et al. [CW13] provide information about an implementation of the Aho-Corasick algorithm in hardware it should be possible to adapt the *SCALCE* algorithm to run on an FPGA. Moreover the boosting is not limited to the implementation of *SCALCE*. *ORCOM* uses a different binning approach but has the same goal to group similar reads and Tan [Tan14] evaluated the application of *K-means clustering* as pre-processing to improve the later compression. A hardware implementation of the *K-means clustering* has already been applied to DNA microarray data and lead to a ten times speed-up compared to a software implementation [Hus+11]. An alternative implementation based on the *filtering* algorithm and kd-trees has been evaluated by Winterstein et al. [WBC13]. The authors showed that their implementation uses about five time less resources than other *K-means clustering* hardware implementations.

In the field of referential compression existing alignment algorithms could be adapted to find matches between the to be compressed sequence and the reference. The task of finding matches for the purpose of compression is insofar easier as any match is suitable[11] whereas during the alignment the best match has to be found. Conversely it is not necessary to match the complete read at once as done during alignment but for compression it is also sufficient to split the read into several smaller chunks and to align them independently. Arram et al. [Arr+13] present an alignment algorithm with an *exact string matcher* based on *FM-index* and an *approximate string matcher* based on a *seed and extend* algorithm. They also mention that for about 70 % to 80 % of the reads exact matches can be expected which might remove the need for the approximate string matches complete if instead a mismatched read is split up to find shorter alignments.

All approaches so far introduced in this section target mainly the compression of nucleotide sequences without considering additional meta-information. To compress for instance the quality scores in FASTQ files the evaluated algorithms used high-order statistical models to achieve good predictions. Although there are implementations of Arithmetic coders for FPGAs they mostly miss the support for complex models and do not provide high compression ratios due to hardware limitations [BNK13]. A commonly used variant is *binary* Arithmetic coding which can be used to emulate *multi-alphabet* coding but also comes to the price of reduced compression ratios.

[11] In a naive implementation matches closer to the beginning of the reference would be beneficial as the number to encode the offset would be smaller. However, this effect can be reduced by using more sophisticated algorithms.

7

Conclusion

The in the scope of this report reviewed algorithms are considered to be the currently best compressors for genetic sequencing data in FASTQ and FASTA format. They feature lossless compression that allows for around six times smaller FASTQ files and twenty times smaller FASTA files while being compliant to different variants of the non standardised formats. Over the last years the compression ratios have mainly been improved by building better models for statistical encoding or by combining different techniques to new compressors. So far the compressors use different models and techniques to compress the different streams of information which are combined in one file. For example the compression of base calls is tackled differently from the compression of the corresponding quality scores. But different structures within each of the streams are not considered although Nevill-Manning et al. [NW99] claim that coding regions do not exhibit Markov properties and are thus incompressible. This would suggest to separate Introns and Exons for the purpose compression. The highly repetitive Introns should be efficiently compressible via Run-length encoding techniques and statistical models whereas Exons might be better compressed via substitutional or referential approaches if they are not easily predictable. However, prior to developing techniques to detect coding and non-coding in a time and memory efficient way the hypothesis of Nevill-Manning et al. [NW99] should be validated by running experiments on known sequences or by calculating the (Shannon) entropy for the different types of regions.

The larger amount of time spent for obtaining the models has lately been compensated by utilising multiple cores. But as most of the well known compression algorithm are in some way inherently sequential the potential benefits in this field are limited. Further, the larger models require enormous amounts of memory and thus cannot be run on everyday desktop computers. A recent alternative is to increase the pipeline depth instead of trying to

parallelise the algorithms. The introduced adaptations of the evaluated algorithms to Dataflow computing are a first step into this direction. Though they seem to match the requirements in theory it has to be verified whether the proposed combinations of algorithms can be fitted into the available resources of an FPGA and whether they lead to performance improvements or at least reduce the energy consumption while providing the same performance.

Regardless of the improvements that might be achieved due to better lossless compression approaches and more efficient hardware utilisation the lossless compression approach seems to reach its limits. For this reason it should be explored which information is applicable for lossy compression without having an impact on the later use of the data. A first approach is made by Illumina Inc. [Ill12] that proposed a projection of the forty currently used quality scores into eight averaged bins. In their evaluation the reduction of the quality scores lead to 26 % smaller FASTQ files without having negative consequences during subsequent analyses such as SNP calling. Further research in the field of lossy as well as lossless quality score transformations has been done by Wan et al. [WAA12] and Yu et al. [Yu+15] found that a quality reduction even improved the accuracy of genotyping. Additionally it should be considered to break the backwards compatibility to the old data formats which have been developed with the aim of being human readable [WAA12]. Being confronted with the huge amounts of data of NGS platforms this is clearly no longer the main objective. Thus a new storage format should be developed and standardised with focus on efficient parsing through computers and better support for compression. This would free the compressors from the need to restore the *exact* variant of the input format during decompression allowing for faster and compacter compression. For instance the read identifiers in the FASTQ format are most of the time identical for all the reads except for a single number indicating the particular read. This can easily be replaced by a per file global identifier for the experiment and the used techniques such that each read is only associated with a single number.

Despite the advances in the compression of the genomic data the files still need large amounts of disk space to be stored and the compressors have high demands for RAM which cannot be afforded by all institutions. In this case the ideas of *Cloud Computing* and *Software as a Service* come to the rescue for small institutions. Instead of transferring and storing all data locally either only the particular part of the datasets (files) needed are accessed and copied or the complete processing is done in high performance data centres and only the results have to be transferred. For instance the random access to parts of the data and easy integration into existing pipeline has been targeted by the frameworks *BEETL* [JSC14], *Goby* [Cam+13] and *SeqDB* [How13].

A

Additional Tables

1st base	2nd base T		2nd base C		2nd base A		2nd base G		3rd base
T	TTT	Phenylalanine (F)	TCT	Serine (S)	TAT	Tyrosine (Y)	TGT	Cysteine (C)	T
	TTC		TCC		TAC		TGC		C
	TTA	Leucine (L)	TCA		TAA	Stop	TGA	Stop	A
	TTG		TCG		TAG	Stop	TGG	Tryptophan (W)	G
C	CTT		CCT	Proline (P)	CAT	Histidine (H)	CGT	Arginine (R)	T
	CTC		CCC		CAC		CGC		C
	CTA		CCA		CAA	Glutamine (Q)	CGA		A
	CTG		CCG		CAG		CGG		G
A	ATT	Isoleucine (I)	ACT	Threonine (T)	AAT	Asparagine (N)	AGT	Serine (S)	T
	ATC		ACC		AAC		AGC		C
	ATA		ACA		AAA	Lysine (K)	AGA	Arginine (R)	A
	ATG	Methionine (M)	ACG		AAG		AGG		G
G	GTT	Valine (V)	GCT	Alanine (A)	GAT	Aspartic acid (D)	GGT	Glycine (G)	T
	GTC		GCC		GAC		GGC		C
	GTA		GCA		GAA	Glutamic acid (E)	GGA		A
	GTG		GCG		GAG		GGG		G

Table A.1 – Overview over all DNA codons. The 64 codons are used to encode the 20 naturally occurring amino acids and three special stop codons. Not all amino acids are encoded by the same number of codons. The RNA codon table can be derived from the DNA codon table by replacing the letter “T” with the letter “U”.

Species	ID	Size on disk [B]	Read count	Read length	Platform	Technique
Arabidopsis thaliana	ERR693886	185 304 212	743 160	96	Illumina	RAD-Seq
	ERR693961	292 842 165	1 173 412	96	Illumina	RAD-Seq
	ERR693963	244 763 046	997 466	94	Illumina	RAD-Seq
	ERR694007	157 888 647	643 590	94	Illumina	RAD-Seq
	ERR694012	331 708 736	1 361 238	93	Illumina	RAD-Seq
Escherichia coli	SRR387476	1 261 116 495	4 908 162	100	Illumina	WGS
	SRR387477	965 784 603	3 759 745	100	Illumina	WGS
	SRR387478	999 501 408	3 890 863	100	Illumina	WGS
	SRR387479	783 311 093	3 050 218	100	Illumina	WGS
	SRR387480	1 148 584 837	4 470 576	100	Illumina	WGS
	SRR387481	145 997 369	570 358	100	Illumina	WGS
	SRR387482	814 506 224	3 171 526	100	Illumina	WGS
	SRR387483	639 866 621	2 492 433	100	Illumina	WGS
	SRR387484	784 008 127	3 052 938	100	Illumina	WGS
Gorilla	SRR957682_1	34 627 357 907	130 354 786	101	Illumina	WGS
	SRR957683_1	26 099 375 362	98 347 626	101	Illumina	WGS
Homo sapiens	ERR039480	15 274 011 931	36 201 289	4–2 716	—	WGS
	ERR039481	14 312 897 731	35 814 702	4–2 876	—	WGS
	ERR039482	11 428 709 276	27 598 278	4–2 716	—	WGS
	ERR039483	15 217 054 813	36 622 655	4–2 716	—	WGS
	ERR039484	13 539 513 164	33 827 169	4–2 716	—	WGS
Mus musculus	ERR850562_1	9 624 047 405	72 177 768	40	Illumina	DNase-Hypersensitivity
	SRR824885	194 321 027	607 381	6–342	—	RNA-Seq
	SRR851145	831 136 761	4 983 535	50	Illumina	ChIP-Seq
	SRR851146	2 028 726 641	12 140 341	50	Illumina	ChIP-Seq
	SRR851147	977 180 486	5 857 365	50	Illumina	ChIP-Seq
Ceratotherium simum	SRR403463_1	46 430 040 046	195 197 750	101	Illumina	WGS
	SRR403850_1	56 165 825 024	235 933 252	101	Illumina	WGS
	SRR409142_1	16 339 244 145	69 035 723	101	Illumina	WGS

Table A.2 – Detailed information over the genetic sequencing data used in the evaluation. The file size relates to the already pre-processed FASTQ files. The data and information have been obtained from `http://www.ebi.ac.uk/` (visited on 21/04/2015).

Compressor	Source code
fastqz	http://mattmahoney.net/dc/fastqz/
fqzcomp	http://sourceforge.net/projects/fqzcomp/
Quip	http://homes.cs.washington.edu/~dcjones/quip/
	https://github.com/dcjones/quip
FRESCO	https://github.com/hubsw/FRESCO[a]
GDC 2	http://sun.aei.polsl.pl/REFRESH/index.php?page=projects&project=gdc&subpage=about[b]
DSRC 2	http://sun.aei.polsl.pl/REFRESH/index.php?page=projects&project=dsrc&subpage=about
	https://github.com/lrog/dsrc
SeqDB	https://bitbucket.org/mhowison/seqdb
ORCOM	http://sun.aei.polsl.pl/REFRESH/index.php?page=projects&project=orcom&subpage=about
	https://github.com/lrog/orcom
SCALCE	http://sfu-compbio.github.io/scalce/
kpath	http://www.cs.cmu.edu/~ckingsf/software/pathenc/

[a] The adapted sources are available at https://gitlab.doc.ic.ac.uk/mp3414/fresco.
[b] The adapted sources are available at https://gitlab.doc.ic.ac.uk/mp3414/gdc.

Table A.3 – List of source code locations for all used specialised compressors. All websites have been visited on 22/04/2015.

B

Additional Code

```
from fractions import Fraction

vertices = {"A" : (Fraction(0), Fraction(0)),
            "C" : (Fraction(0), Fraction(1)),
            "G" : (Fraction(1), Fraction(1)),
            "T" : (Fraction(1), Fraction(0))}

def compress(seq):
    x = y = Fraction(1, 2)

    for c in seq:
        (vx, vy) = vertices.get(c, (x, y))
        x = (x + vx) / 2
        y = (y + vy) / 2

    return (x, y)

def __findRegion(x, y):
    if (x < Fraction(1, 2)) & (y < Fraction(1, 2)):
        return "A"
    elif (x < Fraction(1, 2)) & (y >= Fraction(1, 2)):
        return "C"
    elif (x >= Fraction(1, 2)) & (y < Fraction(1, 2)):
        return "T"
    elif (x >= Fraction(1, 2)) & (y >= Fraction(1, 2)):
        return "G"

def decompress(point):
    (x, y) = point
    seq = ""

    while (x != Fraction(1, 2)) & (y != Fraction(1, 2)):
        symbol = __findRegion(x, y)
        seq = symbol + seq

        (vx, vy) = vertices.get(symbol)

        x = 2 * x - vx
        y = 2 * y - vy

    return seq
```

Code B.1 – Sample implementation of the Chaos Game Representation. This implementation demonstrates the basic principles behind CGR without being further optimised. After compression the coordinates which represent the sequence are returned.

```
import math

def createCompressDictionaryFromAlphabet(alphabet):
    return {alphabet[i] : i for i in range(len(alphabet))}

def createDecompressDictionaryFromAlphabet(alphabet):
    return {i : alphabet[i] for i in range(len(alphabet))}

def compress(uncompressed, dictionary):
    w = ""
    result = []
    dict_size = len(dictionary)
```

```
    for c in uncompressed:
        prefix = w + c

        if prefix in dictionary:
            w = prefix
        else:
            result.append(dictionary[w])
            dictionary[prefix] = dict_size
            dict_size += 1
            w = c

    if w:
        result.append(dictionary[w])

    return (result, math.ceil(math.log2(dict_size)))

def decompress(compressed, dictionary):
    dict_size = len(dictionary)
    result = ""

    w = compressed.pop(0)

    if w in dictionary:
        result += dictionary[w]
        w = dictionary[w]
    else:
        raise ValueError('Bad␣compressed␣symbol:␣%s' % w)

    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError('Bad␣compressed␣symbol:␣%s' % k)

        result += entry

        dictionary[dict_size] = w + entry[0]
        dict_size += 1

        w = entry

    return result
```

Code B.2 – Sample implementation of the Lempel-Ziv-Welch algorithm. This implementation demonstrates the basic principles behind the LZW algorithm without being further optimised. After compression the compressed sequence and the number of bits needed per code symbol are returned.

```
from fractions import Fraction
from math import floor

def createModel(seq):
    assert "\x00" not in seq
    seq += "\x00"
    counts = {}

    for c in seq:
```

```
        counts[c] = counts.get(c, 0) + 1

    return __buildModel(counts)

def __buildModel(counts):
    model = {}
    lower = Fraction(0)
    upper = Fraction(1)
    lettersCount = sum(counts.values())

    for (c, count) in sorted(counts.items(),
                             key=lambda itemCount : itemCount[::-1],
                             reverse=True):
        upper = lower + Fraction(count, lettersCount)
        model[c] = (lower, upper)
        lower = upper

    return model

def compress(seq, model):
    assert "\x00" not in seq
    seq += "\x00"
    lower = Fraction(0)
    upper = Fraction(1)

    for c in seq:
        characterInterval = model[c]

        delta = upper - lower
        upper = lower + characterInterval[1] * delta
        lower = lower + characterInterval[0] * delta

    delta = upper - lower
    bits = 0

    while delta < 1:
        bits += 1
        delta *= 2

    if bits == 0:
        return (lower, upper, 0, Fraction(0))
    else:
        return (lower, upper, bits, ((lower + upper) / 2))

def decompress(value, model):
    seq = ""
    flatModel = [(c, lower, upper)
                 for (c, (lower, upper)) in model.items()]

    while True:
        for (c, lower, upper) in flatModel:
            if lower <= value < upper:
                break
        else:
            raise AssertionError("Not␣valid␣interval!")

        if c == "\x00":
            break

        delta = upper - lower
        value = (value - lower) / delta
```

```
        seq += c

    return seq
```

Code B.3 – Sample implementation of Arithmetic coding. This implementation demonstrates the basic principles behind Arithmetic coding without being further optimised. After compression the final interval representing the sequence and the number of bits needed to represent a number from the final interval are returned.

Bibliography

[AC75] Alfred V. Aho and Margaret J. Corasick. 'Efficient String Matching: An Aid to Bibliographic Search'. In: *Commun. ACM* 18.6 (1975), pp. 333–340. ISSN: 0001-0782. DOI: 10.1145/360825.360855.

[ANO13] K. S. Arun, Achuthsankar S. Nair and Oommen V. Oommen. 'A Novel DNA Sequence Compression Method Based on Chaos Game Representation'. In: *International Journal for Computational Biology (IJCB)* 2.1 (2013), pp. 1–11. ISSN: 2278-8115.

[Ans09] Wilhelm J. Ansorge. 'Next-generation DNA sequencing techniques'. In: *New Biotechnology* 25.4 (2009), pp. 195–203. ISSN: 1871-6784. DOI: 10.1016/j.nbt.2008.12.009.

[Arr+13] James Arram et al. 'Reconfigurable Acceleration of Short Read Mapping'. In: *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*. Apr. 2013, pp. 210–217. DOI: 10.1109/FCCM.2013.57.

[BM13] James K. Bonfield and Matthew V. Mahoney. 'Compression of FASTQ and SAM Format Sequencing Data'. In: *PLoS ONE* 8.3 (2013). ISSN: 19326203. DOI: 10.1371/journal.pone.0059190.

[BNK13] Anton Biasizzo, Franc Novak and Peter Korošec. 'A Multi–Alphabet Arithmetic Coding Hardware Implementation for Small FPGA Devices'. In: *Journal of Electrical Engineering* 64.1 (2013), pp. 44–49. ISSN: 1335-3632. DOI: 10.2478/jee-2013-0006.

[Bon+06] Flavio Bonomi et al. 'An Improved Construction for Counting Bloom Filters'. In: *Algorithms – ESA 2006*. Ed. by Yossi Azar and Thomas Erlebach. Vol. 4168. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 684–695. ISBN: 978-3-540-38875-3. DOI: 10.1007/11841036_61.

[BP14] Kakoli Banerjee and R. A. Prasad. 'A new technique in reference based DNA sequence compression algorithm: Enabling partial decompression'. In: *International Conference of Computational Methods in Sciences and Engineering 2014 (Iccmse 2014)* 799 (2014). ISSN: 0094-243X. DOI: 10.1063/1.4897853.

[BS13] Nour S. Bakr and Amr A. Sharawi. 'DNA Lossless Compression Algorithms: Review'. In: *American Journal of Bioinformatics Research* 3.3 (2013), pp. 72–81. DOI: 10.5923/j.bioinformatics.20130303.04.

[Cam+13] Fabien Campagne et al. 'Compression of structured high-throughput sequencing data'. In: *PLoS ONE* 8.11 (2013). ISSN: 19326203. DOI: 10.1371/journal.pone.0079871.

[CCB12] Guy Cochrane, Charles E Cook and Ewan Birney. 'The future of DNA sequence archiving'. In: *GigaScience* 1.1 (2012), p. 2. ISSN: 2047-217X. DOI: 10.1186/2047-217X-1-2.

[Cha+05] Moses Charikar et al. 'The smallest grammar problem'. In: *IEEE Transactions on Information Theory* 51.7 (2005), pp. 2554–2576. ISSN: 00189448. DOI: 10.1109/TIT.2005.850116.

[CL04] Neva Cherniavsky and Richard Ladner. 'Grammar-based compression of DNA sequences'. In: *DIMACS Working Group on The Burrows-Wheeler Transform* 21 (2004).

[Coc+10] Peter J. A. Cock et al. 'The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants'. In: *Nucleic Acids Research* 38.6 (2010), pp. 1767–1771. DOI: 10.1093/nar/gkp1137.

[Cro+15] Maxime Crochemore et al. 'Computing the Burrows–Wheeler transform in place and in small space'. In: *Journal of Discrete Algorithms* 32.2008217 (2015). ISSN: 15708667. DOI: 10.1016/j.jda.2015.01.004.

[CW08] Wei Cui and Siliang Wu. 'An Improved LZW Data Compression Algorithm and Its VLSI Implementation'. In: *Chinese Journal of Electronics* 17.2 (2008).

[CW13] Chien-Chi Chen and Sheng-De Wang. 'An Efficient Multicharacter Transition String-Matching Engine Based on the Aho-Corasick Algorithm'. In: *ACM Transactions on Architecture and Code Optimization* 10.4 (2013). ISSN: 15443566. DOI: 10.1145/2541228.2541232.

[CW84] John G. Cleary and I. Witten. 'Data Compression Using Adaptive Coding and Partial String Matching'. In: *Communications, IEEE Transactions on* 32.4 (1984), pp. 396–402.

[DDG14] Agnieszka Danek, Sebastian Deorowicz and Szymon Grabowski. 'Indexing large genome collections on a PC'. In: 9.10 (2014), pp. 1–7. ISSN: 19326203. DOI: 10.1371/journal.pone.0109384. URL: http://arxiv.org/abs/1403.7481.

[DDN15] Sebastian Deorowicz, Agnieszka Danek and Marcin Niemiec. 'GDC 2: Compression of large collections of genomes'. In: (2015). arXiv: 1503.01624. URL: http://arxiv.org/abs/1503.01624 (visited on 17/04/2015).

[DG11a] Sebastian Deorowicz and Szymon Grabowski. *Compression of DNA sequence reads in FASTQ format. Supplementary material*. Supplementary Material. 2011.

[DG11b] Sebastian Deorowicz and Szymon Grabowski. 'Compression of DNA sequence reads in FASTQ format'. In: *Bioinformatics* 27.6 (2011), pp. 860–862. ISSN: 13674803. DOI: 10.1093/bioinformatics/btr014.

[DG11c] Sebastian Deorowicz and Szymon Grabowski. 'Robust relative compression of genomes with random access'. In: *Bioinformatics* 27.21 (2011), pp. 2979–2986. ISSN: 13674803. DOI: 10.1093/bioinformatics/btr505.

[DG13] Sebastian Deorowicz and Szymon Grabowski. 'Data compression for sequencing data'. In: *Algorithms for molecular biology : AMB* 8.1 (2013). ISSN: 1748-7188. DOI: 10.1186/1748-7188-8-25.

[DTW05] Richard C. Deonier, Simon Tavaré and Michael Waterman. *Computational Genome Analysis. An Introduction*. 1st ed. Springer-Verlag New York, 2005, p. 535. ISBN: 978-0-387-98785-9. DOI: 10.1007/0-387-28807-4.

[EV14] James A. Edwards and Uzi Vishkin. 'Parallel algorithms for Burrows-Wheeler compression and decompression'. In: *Theoretical Computer Science* 525 (2014), pp. 10–22. ISSN: 03043975. DOI: 10.1016/j.tcs.2013.10.009.

[GA] Jean-loup Gailly and Mark Adler. *GZIP Algorithm*. URL: http://www.gzip.org/algorithm.txt (visited on 13/04/2015).

[GDR14] Szymon Grabowski, Sebastian Deorowicz and Łukasz Roguski. 'Disk-based compression of data from genome sequencing'. In: *Bioinformatics* (2014). DOI: 10.1093/bioinformatics/btu844.

[Gro15] The HDF Group. *Hierarchical Data Format, version 5*. 1997–2015. URL: http://www.hdfgroup.org/HDF5/ (visited on 18/04/2015).

[GRU14] Raffaele Giancarlo, Simona E. Rombo and Filippo Utro. 'Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies'. In: *Briefings in Bioinformatics* 15.3 (2014), pp. 390–406. DOI: 10.1093/bib/bbt088.

[GW13] Ayman Grada and Kate Weinbrecht. 'Next-Generation Sequencing: Methodology and Application'. In: *Journal of Investigative Dermatology* 133.8 (2013). DOI: 10.1038/jid.2013.248.

[Hac+12] Faraz Hach et al. 'SCALCE: boosting sequence compression algorithms using locally consistent encoding'. In: *Bioinformatics* 28.23 (2012), pp. 3051–3057. DOI: 10.1093/bioinformatics/bts593.

[He+12] Ping-an He et al. 'A 3D graphical representation of protein sequences based on the Gray code'. In: *Journal of Theoretical Biology* 304 (2012), pp. 81–87. ISSN: 00225193. DOI: 10.1016/j.jtbi.2012.03.023.

[He+14] Na He et al. 'Multi-task parallel algorithm for DSRC'. In: *Procedia Computer Science* 31 (2014), pp. 1133–1139. ISSN: 18770509. DOI: 10.1016/j.procs.2014.05.369.

[He10] P. He. 'A new graphical representation of similarity/dissimilarity studies of protein sequences'. In: *SAR and QSAR in Environmental Research* 21.5–6 (2010), pp. 571–580. ISSN: 1062-936X. DOI: 10.1080/1062936X.2010.510481.

[HL13] Richard Cg Holland and Nick Lynch. 'Sequence squeeze: an open contest for sequence compression'. In: *GigaScience* 2.1 (2013). ISSN: 2047-217X. DOI: 10.1186/2047-217X-2-5.

[How13] Mark Howison. 'High-throughput compression of FASTQ data with SeqDB'. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 10.1 (2013), pp. 213–218. ISSN: 15455963. DOI: 10.1109/TCBB.2012.160.

[Hus+11] Hanaa M. Hussain et al. 'FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering Microarray data'. In: *Proceedings of the 2011 NASA/ESA Conference on Adaptive Hardware and Systems, AHS 2011* (2011), pp. 248–255. DOI: 10.1109/AHS.2011.5963944.

[Hut07] Clyde A. Hutchison. 'DNA sequencing: bench to bedside and beyond'. In: *Nucleic Acids Research* 35.18 (2007), pp. 6227–6237. DOI: 10.1093/nar/gkm688.

[Ill11] Illumina Inc. *CASAVA v.1.8.2 User Guide*. Version 15011196 Rev D. 2011.

[Ill12] Illumina Inc. *Reducing Whole-Genome Data Storage Footprint*. White Paper 970-2012-013. 2012.

[Jef90] H. J. Jeffrey. 'Chaos game representation of gene structure'. In: *Nucleic acids research* 18.8 (1990), pp. 2163–2170. ISSN: 0305-1048. DOI: 10.1093/nar/18.8.2163.

[Jon+12] Daniel C. Jones et al. 'Compression of next-generation sequencing reads aided by highly efficient de novo assembly'. In: *Nucleic Acids Research* 40.22 (2012), pp. 1–9. ISSN: 03051048. DOI: 10.1093/nar/gks754.

[JSC14] Lilian Janin, Ole Schulz-Trieglaff and Anthony J Cox. 'BEETL-fastq: a searchable compressed archive for DNA reads'. In: *Bioinformatics (Oxford, England)* 30.19 (2014). ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btu387.

[Kah11] Scott D. Kahn. 'On the Future of Genomic Data'. In: *Science* 331.6018 (2011), pp. 728–729. DOI: 10.1126/science.1197891.

[KP15] Carl Kingsford and Rob Patro. 'Reference-based compression of short-read sequences using path encoding'. In: *Bioinformatics* (2015). DOI: 10.1093/bioinformatics/btv071.

[KPZ11] Shanika Kuruppu, Simon J. Puglisi and Justin Zobel. 'Reference sequence construction for relative compression of genomes'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7024 LNCS (2011), pp. 420–425. ISSN: 03029743. DOI: 10.1007/978-3-642-24583-1_41.

[KY00] John C. Kieffer and En Hui Yang. 'Grammar-based codes: a new class of universal lossless source codes'. In: *IEEE Transactions on Information Theory* 46.3 (2000), pp. 737–754. ISSN: 00189448. DOI: 10.1109/18.841160.

[LAS13] E Jebamalar Leavline, D Asir Antony and Gnana Singh. 'Hardware Implementation of LZMA Data Compression Algorithm'. In: *International Journal of Applied Information Systems (IJAIS)* 5.4 (2013), pp. 51–56.

[LC14] N. F. Law and K. O. Cheng. 'A Survey of Techniques for Sequence Similarities Matching in Compression'. In: *Advances in Robotics & Automation* 3.1 (2014). DOI: 10.4172/2168-9695.100011.

[Li+14] Bing Li et al. 'Implementation of LZMA compression algorithm on FPGA'. In: *Electronics Letters* 50.21 (2014), pp. 1522–1524. ISSN: 0013-5194. DOI: 10.1049/el.2014.1734.

[Mah14] Matt Mahoney. *The ZPAQ Open Standard Format for Highly Compressed Data – Level 2*. Open Standard. Version 2.04. Dell Inc., 2014. URL: http://mattmahoney.net/dc/zpaq204.pdf (visited on 13/04/2015).

[Mar08] Elaine R. Mardis. 'The impact of next-generation sequencing technology on genetics'. In: *Trends in Genetics* 24.3 (2008), pp. 133–141. ISSN: 0168-9525. DOI: 10.1016/j.tig.2007.12.007.

[Mar79] G. N. N. Martin. *Range encoding: an algorithm for removing redundancy from a digitised message*. Conference Paper. Southampton: IBM UK Scientific Center, 1979.

[Mas+10] ChristopherE. Mason et al. 'Standardizing the Next Generation of Bioinformatics Software Development with BioHDF (HDF5)'. In: *Advances in Computational Biology*. Ed. by Hamid R. Arabnia. Vol. 680. Advances in Experimental Medicine and Biology. Springer New York, 2010, pp. 693–700. ISBN: 978-1-4419-5912-6. DOI: 10.1007/978-1-4419-5913-3_77.

[Mee14] Mary Meeker. 'Internet Trends 2014 – Code Conference'. In: 2014. URL: http://s3.amazonaws.com/kpcbweb/files/85/Internet%5C_Trends%5C_2014%5C_vFINAL%5C_-%5C_05%5C_28%5C_14-%5C_PDF.pdf?1401286773 (visited on 31/03/2015).

[MG77] Allan M. Maxam and Walter Gilbert. 'A new method for sequencing DNA'. In: *Proceedings of the National Academy of Sciences* 74.2 (1977), pp. 560–564.

[MSI00] T. Matsumoto, K. Sadakane and H. Imai. 'Biological sequence compression algorithms'. In: *Genome Inform Ser Workshop Genome Inform* 11 (2000), pp. 43–52.

[NCB07] National Center for Biotechnology Information. *Query Input and database selection*. 2007. URL: http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml (visited on 01/04/2015).

[Now10] Minou Nowrousian. 'Next-Generation Sequencing Techniques for Eukaryotic Microorganisms: Sequencing-Based Solutions to Biological Problems'. In: *Eukaryotic Cell* 9.9 (2010), pp. 1300–1310. DOI: 10.1128/EC.00123-10.

[NW99] C. G. Nevill-Manning and I. H. Witten. 'Protein is incompressible'. In: *Proceedings DCC'99 Data Compression Conference (Cat. No. PR00096)* (1999). ISSN: 1068-0314. DOI: 10.1109/DCC.1999.755675.

[PS08] Mihai Pop and Steven L. Salzberg. 'Bioinformatics challenges of new sequencing technology'. In: *Trends in Genetics* 24.3 (2008), pp. 142–149. DOI: 10.1016/j.tig.2007.12.006.

[PS11] Pavel Pevzner and Ron Shamir, eds. *Bioinformatics for Biologists*. Cambridge University Press, 2011. ISBN: 9780511984570. DOI: 10.1017/CBO9780511984570.

[RBK07] Suzanne Rigler, William Bishop and Andrew Kennings. 'FPGA-based lossless data compression using Huffman and LZ77 algorithms'. In: *Canadian Conference on Electrical and Computer Engineering* (2007), pp. 1235–1238. ISSN: 08407789. DOI: 10.1109/CCECE.2007.315.

[RD14a] Łukasz Roguski and Sebastian Deorowicz. *DSRC 2 – Industry-oriented compression of FASTQ files. Supplementary material*. Supplementary Material. 2014.

[RD14b] Łukasz Roguski and Sebastian Deorowicz. 'DSRC 2 – Industry-oriented compression of FASTQ files'. In: *Bioinformatics* 30.15 (2014), pp. 2213–2215. ISSN: 1367-4803, 1460-2059. DOI: 10.1093/bioinformatics/btu208.

[Rei09] Jorge S. Reis-Filho. 'Next-generation sequencing'. In: *Breast Cancer Research* 11.Suppl 3 (2009). DOI: 10.1186/bcr2431.

[RFC1951] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. Request for Comments. Network Working Group, May 1996. URL: http://www.ietf.org/rfc/rfc1951.txt (visited on 13/04/2015).

[RFC1952] P. Deutsch. *GZIP file format specification version 4.3*. Request for Comments. Network Working Group, May 1996. URL: http://www.ietf.org/rfc/rfc1952.txt (visited on 13/04/2015).

[Rig07] Suzanne Rigler. 'FPGA-Based Lossless Data Compression Using GNU Zip'. Master's thesis. University of Waterloo, 2007.

[RK15] Strahil Ristov and Damir Korenčić. 'Using static suffix array in dynamic application: Case of text compression by longest first substitution'. In: *Information Processing Letters* 115 (2015), pp. 175–181. ISSN: 00200190. DOI: 10.1016/j.ipl.2014.08.014.

[Rob+04] Michael Roberts et al. 'Reducing storage requirements for biological sequence comparison'. In: *Bioinformatics* 20.18 (2004), pp. 3363–3369. DOI: 10.1093/bioinformatics/bth408.

[SC75] Frederick Sanger and Alan R. Coulson. 'A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase'. In: *Journal of Molecular Biology* 94.3 (1975), pp. 441–448. ISSN: 0022-2836. DOI: 10.1016/0022-2836(75)90213-2.

[Sew07] Julian Seward. *bzip2 and libbzip2. A program and library for data compression*. Version 1.0.5. 2007. URL: http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.pdf (visited on 13/04/2015).

[SFS15a] The SAM/BAM Format Specification Working Group. *CRAM format specification*. Tech. rep. Version 3. 2015. URL: http://samtools.github.io/hts-specs/CRAMv3.pdf (visited on 01/04/2015).

[SFS15b] The SAM/BAM Format Specification Working Group. *Sequence Alignment/Map Format Specification*. Tech. rep. Version 1. 2015. URL: http://samtools.github.io/hts-specs/SAMv1.pdf (visited on 01/04/2015).

[Shk02] Dmitry Shkarin. 'PPM: one step to practicality'. In: *Data Compression Conference, 2002. Proceedings. DCC 2002*. 2002, pp. 202–211. DOI: 10.1109/DCC.2002.999958.

[SNC77] Frederick Sanger, S. Nicklen and Alan R. Coulson. 'DNA sequencing with chain-terminating inhibitors'. In: *Proceedings of the National Academy of Sciences of the United States of America* 74.12 (1977), pp. 5463–5467. ISSN: 0027-8424. DOI: 10.1073/pnas.74.12.5463.

[SS82] James A. Storer and Thomas G. Szymanski. 'Data Compression via Textual Substitution'. In: *J. ACM* 29.4 (Oct. 1982), pp. 928–951. ISSN: 0004-5411. DOI: 10.1145/322344.322346.

[SV96] S.C. Sahinalp and U. Vishkin. 'Efficient approximate and dynamic matching of patterns using a labeling paradigm'. In: *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*. Oct. 1996, pp. 320–328. DOI: 10.1109/SFCS.1996.548491.

[Tan14] Li Tan. 'K-means Clustering Based Compression Algorithm for the High-throughput DNA Sequence'. In: (2014), pp. 952–955.

[Vid+11] S. M. Vidanagamachchi et al. 'Tile optimization for area in FPGA based hardware acceleration of peptide identification'. In: *Industrial and Information Systems (ICIIS), 2011 6th IEEE International Conference on*. Aug. 2011, pp. 140–145. DOI: 10.1109/ICIINFS.2011.6038056.

[WAA12] Raymond Wan, Vo Ngoc Anh and Kiyoshi Asai. 'Transformations for the compression of FASTQ quality scores of next-generation sequencing data'. In: *Bioinformatics* 28.5 (2012), pp. 628–635. ISSN: 13674803. DOI: 10.1093/bioinformatics/btr689.

[WBC13] F. Winterstein, S. Bayliss and G. A. Constantinides. 'FPGA-based K-means clustering using tree-based data structures'. In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. Sept. 2013. DOI: 10.1109/FPL.2013.6645501.

[WBL13] Sebastian Wandelt, Marc Bux and Ulf Leser. 'Trends in Genome Compression'. In: *Current Bioinformatics* 9 (2013). ISSN: 15748936.

[Wel84] Terry A. Welch. 'Technique for High-Performance Data Compression'. In: *Computer* 17.6 (1984), pp. 8–19. ISSN: 00189162. DOI: 10.1109/MC.1984.1659158.

[WikiBzip2] *Bzip2 — Wikipedia, The Free Encyclopedia*. Wikipedia. 2015. URL: http://en.wikipedia.org/w/index.php?title=Bzip2%5C&oldid=656033049 (visited on 13/04/2015).

[WL13] Sebastian Wandelt and Ulf Leser. 'FRESCO: Referential Compression of Highly-Similar Sequences'. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 10.5 (2013). ISSN: 1545-5963. DOI: 10.1109/TCBB.2013.122.

[Yu+15] Y William Yu et al. 'Quality score compression improves genotyping accuracy'. In: *Nature Biotechnology* 33.3 (2015), pp. 240–243. ISSN: 1087-0156. DOI: 10.1038/nbt.3170.

[ZL77] J. Ziv and A. Lempel. 'A universal algorithm for sequential data compression'. In: *IEEE Transactions on Information Theory* 23.3 (1977). ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

[ZL78] J. Ziv and A. Lempel. 'Compression of individual sequences via variable-rate coding'. In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055934.