



**UNIVERSIDAD  
DE ANTIOQUIA**

1 8 0 3

# **Efficient Storage of Genomic Sequences in High Performance Computing Systems**

**Aníbal José Guerra Soler**

Facultad de Ingeniería  
Universidad de Antioquia  
Medellín, Colombia  
Mayo, 2019



# **Efficient Storage of Genomic Sequences in High Performance Computing Systems**

**Aníbal José Guerra Soler**

Tesis para optar al título de:  
**Doctor en Ingeniería Electrónica y de Computación**

Directores:

Prof. Sebastián Isaza Ramírez, PhD.

Prof. José Édinson Aedo Cobo, PhD.

Grupo de Investigación:  
Sistemas Embebidos e Inteligencia Computacional - SISTEMIC

Facultad de Ingeniería  
Universidad de Antioquia  
Medellín, Colombia  
Mayo 2019



## Abstract

In this dissertation, we address the challenges of genomic data storage in high performance computing systems. In particular, we focus on developing a referential compression approach for Next Generation Sequence data stored in FASTQ format files. The amount of genomic data available for researchers to process has increased exponentially, bringing enormous challenges for its efficient storage and transmission. General-purpose compressors can only offer limited performance for genomic data, thus the need for specialized compression solutions. Two trends have emerged as alternatives to harness the particular properties of genomic data: non-referential and referential compression. Non-referential compressors offer higher compression ratios than general purpose compressors, but still below of what a referential compressor could theoretically achieve. However, the effectiveness of referential compression depends on selecting a good reference and on having enough computing resources available. This thesis presents one of the first referential compressors for FASTQ files. We first present a comprehensive analytical and experimental evaluation of the most relevant tools for genomic raw data compression, which led us to identify the main needs and opportunities in this field. As a consequence, we propose a novel compression workflow that aims at improving the usability of referential compressors. Subsequently, we discuss the implementation and performance evaluation for the core of the proposed workflow: a referential compressor for reads in FASTQ format that combines local read-to-reference alignments with a specialized binary-encoding strategy. The compression algorithm, named UdeACompress, achieved very competitive compression ratios when compared to the best compressors in the current state of the art, while showing reasonable execution times and memory use. In particular, UdeACompress outperformed all competitors when compressing long reads, typical of the newest sequencing technologies. Finally, we study the main aspects of the data-level parallelism in the Intel AVX-512 architecture, in order to develop a parallel version of the UdeACompress algorithms to reduce the runtime. Through the use of SIMD programming, we managed to significantly accelerate the main bottleneck found in UdeACompress, the Suffix Array Construction.

**Keywords:** Reads Compression, Referential compression, Reads alignment, Genomic sequences, Parallel computing, SIMD programming, Performance evaluation.

*To God and my parents...*

# Contents

<b>1. Introduction</b>	<b>1</b>
1.1. Motivation . . . . .	1
1.2. Thesis Contributions . . . . .	3
1.3. Thesis Organization . . . . .	4
<b>2. Background</b>	<b>6</b>
2.1. Compression Techniques . . . . .	6
2.1.1. Genomic Data . . . . .	6
2.1.2. Trends in Compression of Genomic Data . . . . .	8
2.1.3. Non-Referential Algorithms . . . . .	10
2.1.4. Referential Algorithms . . . . .	12
2.2. High Performance Computing Systems . . . . .	13
2.3. Parallel Programming . . . . .	14
2.3.1. SIMD Level Parallelism . . . . .	15
2.3.2. Thread Level Parallelism . . . . .	16
2.4. Summary . . . . .	17
<b>3. Evaluation of the State of the Art in Genomic Data Compression</b>	<b>18</b>
3.1. State of the Art and Performance Evaluation of DNA compressors . . . . .	19
3.1.1. Evaluated Compressors . . . . .	20
3.1.2. Experimental Methodology . . . . .	26
3.1.3. Results and Discussion . . . . .	27
3.2. State of the Art on Referential Compressors . . . . .	37
3.2.1. Genome Data Compression . . . . .	37
3.2.2. Compression of Read Sequences Along with Alignment/Mapping Information . . . . .	38
3.2.3. Compression of Next Generation Sequencing Raw Data . . . . .	38
3.3. Summary . . . . .	40
<b>4. A Workflow for Referential Compression of FASTQ</b>	<b>42</b>
4.1. The Workflow Model . . . . .	42
4.2. Sequences Database . . . . .	44
4.3. Packing and Unpacking . . . . .	45
4.4. Sequences Features Detection . . . . .	45

---

4.5. Read Sequences Classification . . . . .	47
4.6. Multi-Technique Compression Scheme . . . . .	47
4.6.1. Lossless Compression of Read Sequences . . . . .	48
4.6.2. Identifiers Compression . . . . .	49
4.6.3. Quality Scores Compression . . . . .	50
4.7. Summary . . . . .	51
<b>5. UdeACompress: A Referential Compressor for FASTQ</b>	<b>53</b>
5.1. Read-Sequences Compression with UdeACompress . . . . .	53
5.1.1. Read-to-Reference Alignment . . . . .	54
5.1.2. Reads Sorting . . . . .	55
5.1.3. Encoding . . . . .	57
5.1.4. Low Level Compression . . . . .	63
5.2. Performance Evaluation . . . . .	64
5.2.1. Real Datasets Tests . . . . .	64
5.2.2. Simulated Datasets Tests . . . . .	73
5.3. Summary . . . . .	78
<b>6. Improving the Execution Speed Through Parallelism</b>	<b>81</b>
6.1. Accelerating the Alignment Algorithm . . . . .	81
6.1.1. Profiling the Seed Algorithm . . . . .	82
6.1.2. Assessing the Parallelization Feasibility in SAC . . . . .	85
6.1.3. SIMD SAC Design . . . . .	86
6.2. Performance Evaluation . . . . .	96
6.2.1. SIMD SAC Performance . . . . .	97
6.2.2. Considerations for Instruction-set Extensions . . . . .	100
6.2.3. Introducing Thread Level Paralellism into the SIMD Approach . . . . .	101
6.3. Summary . . . . .	102
<b>7. Conclusions and Further Research</b>	<b>104</b>
<b>A. FASTQ Compression Results</b>	<b>110</b>
<b>B. FASTQ Read Sequences Compression</b>	<b>111</b>



# List of Figures

<b>1-1.</b> Comparison between changes in price of sequencing and price of storage storage in the last decades. . . . .	2
<b>2-1.</b> Structure of a DNA double helix . . . . .	7
<b>2-2.</b> Basic structure of the FASTQ format . . . . .	8
<b>2-3.</b> Basic compression methods . . . . .	10
<b>2-4.</b> Basic compression methods . . . . .	12
<b>2-5.</b> Referential compression . . . . .	12
<b>2-6.</b> Multiprocessors Architecture . . . . .	13
<b>2-7.</b> Evolution of Intel’s SIMD extension on top of x86/x87. . . . .	16
<b>3-1.</b> Compression ratio of evaluated programs . . . . .	27
<b>3-2.</b> Single-thread throughput . . . . .	28
<b>3-3.</b> Multi-threaded throughput . . . . .	30
<b>3-4.</b> Speedup . . . . .	31
<b>3-5.</b> Maximum peak memory during single-thread execution . . . . .	32
<b>3-6.</b> Maximum average memory in single-thread execution . . . . .	33
<b>3-7.</b> Peak memory consumption . . . . .	34
<b>3-8.</b> Average memory consumption . . . . .	34
<b>4-1.</b> Referential compression workflow . . . . .	43
<b>4-2.</b> Multi-technique compression scheme . . . . .	48
<b>5-1.</b> UdeACompress block diagram . . . . .	54
<b>5-2.</b> Instruction encoding . . . . .	57
<b>5-3.</b> Circular base distances scheme. . . . .	60
<b>5-4.</b> Example of instruction encoding to represent the three reads shown . . . . .	62
<b>5-5.</b> Profile of the sequential version of UdeACompress . . . . .	66
<b>5-6.</b> Compression ratio for FASTQ files. . . . .	68
<b>5-7.</b> Compression ratio for the read sequences . . . . .	69
<b>5-8.</b> Throughput during compression of FASTQ files . . . . .	70
<b>5-9.</b> Throughput during compression of the read sequences . . . . .	71
<b>5-10.</b> Throughput during decompression of FASTQ files. . . . .	71
<b>5-11.</b> Throughput during decompression of the read sequences. . . . .	72

---

<b>5-12.</b> Compression ratio while varying the read lengths . . . . .	76
<b>5-13.</b> Compression ratio while varying the percentage of mutations . . . . .	77
<b>5-14.</b> Compression ratio while varying coverage . . . . .	78
<b>6-1.</b> Stages of the seeding phase . . . . .	82
<b>6-2.</b> Call-graph of the process of building the FM-Index . . . . .	83
<b>6-3.</b> Example of calculating the BWM(T) . . . . .	84
<b>6-4.</b> Comparison of the BWM(T) and the SA . . . . .	84
<b>6-5.</b> Process of the suffix array construction . . . . .	85
<b>6-6.</b> Multi-level parallel Suffix Array Construction . . . . .	88
<b>6-7.</b> Flowchart of the PartialGroupSort . . . . .	89
<b>6-8.</b> Flowchart of the SIMD Suffix Array Construction Algorithm . . . . .	89
<b>6-9.</b> SAC execution time . . . . .	97
<b>6-10.</b> Profiling of both versions of the SIMD SAC algorithm. . . . .	98
<b>6-11.</b> Speedup of SIMD_Naive and SIMD_OPT against Scalar. . . . .	100

# List of Tables

<b>3-1.</b>	Tools configuration used during experiments. . . . .	21
<b>3-2.</b>	Effect of best configuration in performance . . . . .	29
<b>3-3.</b>	Summary of best multi-threaded performance results . . . . .	31
<b>4-1.</b>	Illumina's identifiers structure . . . . .	49
<b>4-2.</b>	Quip and DSRC compression ratio on FASTQ IDs and Qs . . . . .	51
<b>5-1.</b>	Matching codes . . . . .	59
<b>5-2.</b>	Bases Complement . . . . .	59
<b>5-3.</b>	Probability of mutations occurrence . . . . .	61
<b>5-4.</b>	Penalty categories . . . . .	62
<b>5-5.</b>	Dataset description . . . . .	65
<b>5-6.</b>	Compression software description . . . . .	66
<b>5-7.</b>	Peak memory consumption during compression and decompression . . . . .	73
<b>5-8.</b>	Simulated tests configuration . . . . .	74
<b>A-1.</b>	Summary of performance results . . . . .	110
<b>B-1.</b>	Summary of performance results . . . . .	111

# List of Algorithms

1.	Reads Sorter . . . . .	56
2.	Instruction Encoder . . . . .	58
3.	Input Chars Generator . . . . .	91
4.	Vector Last Unique . . . . .	92
5.	Vector Prior Instances . . . . .	93
6.	Partial Radixsort . . . . .	94

## Acknowledgements

First of all, to the most important person in my life; without whom I am nothing, my God and savior: Jesus. To the beautiful Colombia, the people of Antioquia and the University of Antioquia for giving me this great opportunity. To my mom, my dad and my aunt Sandra, for the great effort they made to educate us; we owe you everything we are. To my sisters and my family, especially the Soler family, your support and care gave me the strength to go on the hard day. To Carmen Ochoa, the pastors Claudia and Diego Montilla, and to all the people that during my life have fed me spiritually; for their support, their prayers and guidance. To my friends: Erika Eschorcha, the Mendoza family, Winder, María Bernarda Salazar, Dinarle Ortega, Joel Rivas, Pan de Avena, Berta, Mariogly, María Gabriela López, María Cuartas, “los españoletes” Xavi and Remé, Marlyn, Paola, María José and the Mexican “súper parcero” Carlos. There is no greater treasure than friendship. To Hanen Hanna at FaCyT for her great support as the “asuntos profesoriales” director. To all those who supported me with encouragement or contributed to my crowdfunding campaign at that very difficult time. To Jaci Velasquez, for writing “confío”, a personal inspiration to me through the darkest times. To Professor Sebastián Isaza, words are missing to thank you. Keep demonstrating that you can be a respected researcher and an excellent person simultaneously. I am infinitely grateful for all the trust, support and even the therapy sessions; I deeply admire the human you are. Infinite thanks for giving me the honor of teaching at UdeA. I wish you the best things. I hope to lose a great tutor today to win a great friend. To the team of students that joined us in these years of work, especially to Jaime and Juan David for going far beyond the simple fulfillment of their work. To all the staff of the engineering postgraduate office for their diligence and collaboration (even those who are no longer there). Especially Professor Natalia Gaviria, for supporting me along this journey, I am sure that without her constant help and encouragement the multitude of procedures would have been a complete nightmare. To Ismenia, always attentive and very diligent, thank you for your cooperation. To Professor Mauricio Hernández, the first who opened the UdeA doors for me, thank you. To COLCIENCIAS for funding part of my studies. To all the teachers and colleagues of SISTEMIC, for their cordiality, support and human warmth. To Germán, for being an excellent laboratory partner, the miscellaneous conversations will be missed. To Professor Felipe Cabarcas, for making me forget the worries with his morning madness. To the professors of the Department of Electronics for their kindness, and to Marina for being always diligent and kind. For their collaboration during the process to Prof. José Aedo and Juan Alzate from CNSG. To Friman Sánchez, Miquele and the staff of the BSC-CNS, for opening the doors of such a prestigious institution in the most kind way. Thank you Friman for the continuous support and the hours invested throughout my learning process. Thank you very much to all of you, I pray you receive a thousand times more than what you have given to me;

**Anibal,**

# 1. Introduction

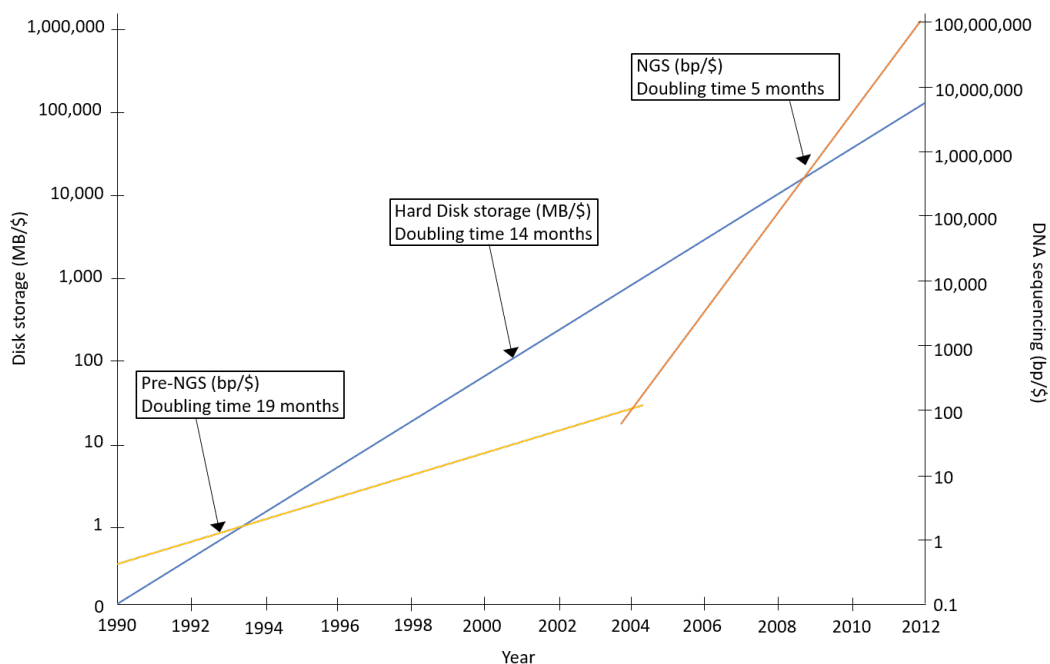
A DNA sequencing run produces raw data that is used for assembling the fundamental data object in bioinformatics: genomes. Genomes are the macromolecule that encodes genetic information through chromosomes, represented as combinations of four biological bases: A (Adenine), C (Cytosine), G (Guanine), and T (Thymine). The understanding of the genome structure and function is essential to many fields in the life sciences, mainly medicine and biology. Recent technological advances (i.e massive parallel processing) have led to the development of Next Generation Sequencing (NGS) technology. Such paradigm change, has allowed to accelerate the drop in sequencing costs. Consequently, an increasing number of genomic studies are being carried out, causing an exponential growth of the amount of NGS raw data available. Such data usually needs to be kept for further studies (mapping, assembling, among others), using computer storage systems that therefore quickly saturate. Data compression is an effective way to overcome this problem, however, it requires enough computing resources and specialized approaches in order to be effective.

This thesis focuses on the development of a referential compressor for NGS raw data in FASTQ format (the de-facto standard). The insights and knowledge obtained from an extensive review of the state of the art, led us to identify the main needs to be tackled through the proposed algorithm. Finally, we studied strategies for improving the algorithm's performance by means of parallel computing.

## 1.1. Motivation

Bioinformatics has driven an accelerated growth of the life sciences, providing technological resources to improve slow and inaccurate traditional wet-lab procedures. One of the fields that has mostly benefited by such advances is genetics, in part due to the rapid improvements in sequencing technology. Through the evolution of the bioinformatics discipline, the sequencing process has not only been greatly accelerated in orders of magnitude, but also its costs have been significantly reduced over the last decade [1–5], even faster than the reduction of storage costs. This phenomenon is evidenced in the logarithmic plot of Figure 1-1 taken from [6], where the blue line represents disk prices in megabytes per US dollar; yellow and red lines represent cost of DNA sequencing in base pair per dollar (base pair is a unit used referred to representing a pair of complementary dna bases, as they appear in the DNA double helix). The intersection between both lines in 2004 marks the point from when DNA sequencing costs began dropping at a much faster rate than of

storage devices. Nonetheless, it is still much cheaper to store the data of a whole genome than re-sequencing it, as sequencing costs per genome start at around of 1000 USD [7], compared to approximately 7.15 USD per year for its data center storage [8].



**Figure 1-1.** Comparison between change in price of sequencing and storage devices. [6]

This reduction in costs has allowed researchers to undertake ambitious projects [9–11] to study an unprecedented amount of individuals. A product of these increasingly large studies is the accelerated growth of genomic databases [6, 12], which comes at accordingly higher costs of storage infrastructure. Additionally, for every megabyte of assembled genomic data, two or more megabytes of NGS raw data are produced. From large research centers to smaller bioinformatics labs, efficient storage for the ever increasing data is one of their major IT challenges [13] in the field.

FASTQ [14] has become the *de-facto* standard file formats in the bioinformatics domain for storing NGS raw data, which is the most important issue of genomic data storage. Today’s NGS machines produce FASTQ files typically containing millions of DNA fragments called *reads* [15]. Given that FASTQ files are stored as plain text, one can easily rely on traditional general-purpose compression tools to compress them. Converting characters into bit streams [16], using static or dynamic dictionaries [17–19] and performing statistical analyses [20] are strategies implemented by many of these tools that alleviate to some extent the problem. These type of tools have seen widespread use for compressing biological sequences [21, 22] due to their compatibility, robustness and ease of use; although not without certain performance limitations [23–25]. However, general purpose compression tools have shown far from optimal performance when applied to the compression of

genomic data [12], due to the nature of these tools, which are optimized for compressing texts in English language or other types of data, such as audio and video. Genomic information differs greatly from these types of data, since not only the alphabet is much more reduced but also the redundancy distributions exhibit a different behavior [26].

In the mean time, domain-specific lossless compressors have been developed during the last decade in an effort to increase efficiency combining traditional techniques with more complex approaches [27–29], to be explained in Chapter 2 of this thesis. However, compressing biological sequences is an intensive task which demands significant computational resources [30]. Two different approaches have led this trend [12, 16]: non-referential and referential compressors. Non-referential compressors [13, 27, 31–38] are commonly easy to use and produce self contained files, but tend to show modest compression ratios. Referential compressors [4, 31, 39–42] may demand a more experienced user, but are able to reach higher compression ratios when using a highly similar reference. Despite their potential, referential compressors are not widely used for two reasons: the need for a reference and the high computational requirements. Furthermore, the achievable compression ratio generally depends on selecting an appropriate reference for every different input. Although promising results have been reported in overall, there are no standard compressors in this field due to several reasons: excessive runtime, difficulty of use, special hardware requirements, lack of essential features (as handling long reads, lossless compression, parallelism support, among others), necessity of reference genomes or particular conditions in which some techniques can be applied [16]. Although at the time of writing this thesis the situation had started to changed, when we did the first state of the art review (2014), we did not find any work on referential compression of NGS raw data. Specialized compression for DNA raw data faces therefore challenges in order to achieve ease of use, higher compression ratios and efficient execution times.

## 1.2. Thesis Contributions

The main objectives of this thesis were set as follows:

- To review and evaluate the state of the art in the field of storage and compression of genomic sequences in high performance computing architectures.
- To build a data bank collecting, preprocessing and documenting genomic data.
- To implement and evaluate different techniques for efficient compression of genomic data.
- To apply parallel processing strategies to accelerate the performance of the algorithms developed.
- To test and evaluate the performance of the developed algorithms using statistical tests.

Finally, the main contributions presented in this thesis are:



- The characterization of the problem through an extensive literature review regarding compression algorithms for genomic data compression. Since the evolution of the compressors for genomic data was rather incipient, we performed a deeper study involving an experimental performance evaluation to identify the features and needs of the compressors in the state of the art. The main insight provided is the comparison of the various techniques and approaches of specialized compressors, measuring their efficiency, performance and scalability.
- The design of a workflow which aims at automatically selecting the reference needed by the compressor, reducing the human intervention required and enhancing the usability of such compression approach. Even though a full implementation is not provided, we show in detail that implementing such idea is promising and provide implementations of some of the most important modules.
- The design and implementation of a Multi-Technique Compression (MTC) scheme which allows to perform a specialized lossless compression that harness the specific features in each of the three streams inside a FASTQ file. The core of this MTC scheme is a referential compressor for the read sequences stream, which we call UdeACompress. It is based on sequence alignment and an elaborate binary encoding scheme, aiming at improving the reads compression ratio for the increasing reads length of newer sequencing machines. We evaluate and compare its performance (compression ratio and speed of execution) on a set of real and simulated data from different organisms and in a set of artificial data which allows to compress much longer reads. Several experiments and metrics are reported and analyzed to conclude about the pros and cons of our proposal.
- A low-level parallelization study of the main computational bottleneck in UdeACompress. After identifying and analyzing the most consuming task in our compressor, we propose SIMD algorithms for the construction of Suffix Arrays. We study, implement and test a set of optimizations that achieved significant performance improvements. Finally, we implement and discuss the combination of the SIMD approach with multi-threading.

### 1.3. Thesis Organization

This manuscript is divided into seven chapters. Chapter 2 contains the main background concepts related to this thesis, in the fields of bioinformatics, compression techniques and high performance computing.

---

Chapter 3 presents a comprehensive review on the current state of the art of the genomic data compression algorithms as well as a detailed survey on the most relevant tools for the compression of FASTQ files. A performance evaluation of the top compressors in this field is also included there. Also, an in depth review of the different approaches applied for referential compression of other types of genomic data is presented.

Chapter 4 introduces the design of a novel workflow to complement the referential compression of FASTQ files. Such workflow describes the sequence of processes through which the required appropriate reference can be automatically selected. The main blocks for each process in the workflow are explained along with the most relevant approaches to achieve the corresponding objectives.

Chapter 5 presents the implementation and evaluation of the workflow's core: UdeACompress. The referential approach is discussed, and experimental results are presented in detail .

Chapter 6 details the efforts for accelerating the main bottleneck found in UdeACompress: the aligner. A detailed evaluation of the inner algorithms of such task are presented, which led to the development of SIMD algorithms to accelerate Suffix Array Construction. Performance results are presented as well as considerations for employing in the future a multi-threaded approach.

Finally, Chapter 7 summarizes the work done, the results achieved and draws the most important conclusions of the thesis. Finally we discuss the future directions of this research.

## 2. Background

As discussed in the previous chapter, genomics databases have been growing exponentially over the last years due to decreasing costs in genome sequencing. The biggest portion of information stored in such databases is the direct product of genome sequencing: the reads and related meta-data.

The development of specialized compression algorithms for NGS raw data is an issue that has received great attention in the last decade, delivering programs with modest compression ratios and exhibiting high execution times. In this chapter, we present the basic concepts related to genomic data and data compression. Also, we discuss the principles of high performance computing, since this type of architectures are commonly available at bioinformatics research centers and could be harnessed in the process of compressing genomic data.

### 2.1. Compression Techniques

In this section we introduce the main concepts related to genomic data and its current storage. Also we present the basis of genomic data compression with a brief emphasis in the compression techniques applied in the referential and non-referential approaches.

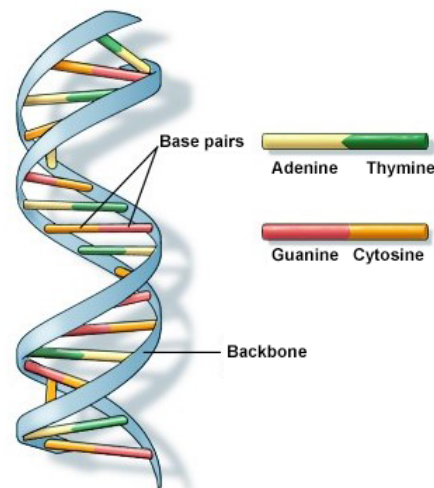
#### 2.1.1. Genomic Data

DNA, or deoxyribonucleic acid, is the hereditary material in humans and all other organisms except for some types of viruses. The information in DNA is stored as a code made up of four chemical bases: adenine (A), guanine (G), cytosine (C), and thymine (T). Human DNA consists of about  $3 \times 10^9$  bases, and more than 99.9% of those bases are the same in all individuals. The order, or sequence, of these bases determines the information available for building and maintaining an organism, similar to the way in which letters of the alphabet appear in a certain order to form words and sentences [43].

DNA bases pair up with each other, A with T and C with G, to form units called base pairs. Each base is also attached to a sugar molecule and a phosphate molecule. Together, a base, sugar, and phosphate are called a nucleotide. Nucleotides are arranged in two long strands that form a spiral

called a double helix. Figure 2-1 shows an illustration of the mentioned structures [43].

The term genome refers to the sum of the DNA contained in an organism's chromosomes. Human DNA comprises the sequence contained in the 23 pairs of chromosomes, with a length of approximately 3.000 million nucleotides. Each nucleotide is described by a letter: guanine (G), adenine (A), thymine (T), or cytosine (C) [44] [15]] (Figure 2-1). As the base pairs are grouped statically (A, T) and (G, C), it is sufficient to represent only one base. Hence in bioinformatics, DNA sequences are abstracted as just sequences of letters, each corresponding to a nucleotide (also called base). However, the importance and complexity of the DNA is major, being the center of life and evolution. DNA carries, in genes, the instructions for making a specific protein or set of proteins. Proteins make up body structures like organs and tissue, as well as control chemical reactions and carry signals between cells. The discovery and understanding of DNA has impacted the development of medicine, agriculture, forensic sciences, laws and many other important fields.



**Figure 2-1.** Structure of a DNA double helix [43]

Data sequences come from laboratory procedures. Most of the commercial NGS systems<sup>1 2 3 4</sup> are based on the in vitro cloning approach. Through biochemical processes, a DNA sample is replicated, fragmented and represented in a matrix structure, then it is transformed into a series of four distinct fluorescent signals (each representing a different base) monitored by a CCD camera. The series of fluorescent signals at each position are converted into a sequence of letters. Differences in the sequencing chemistry of each NGS platform results in differences in total sequence capacity, sequence read length, sequence run time, and final quality and accuracy of the data. A

<sup>1</sup><https://sequencing.roche.com/en.html>

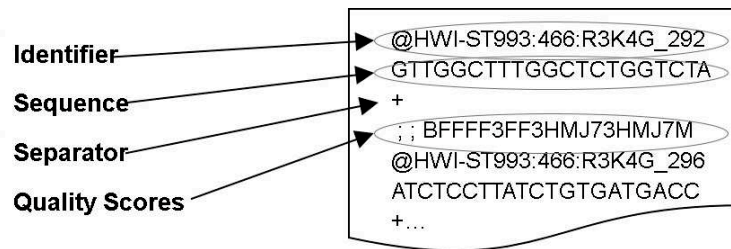
<sup>2</sup><https://www.thermofisher.com>

<sup>3</sup><https://www.illumina.com/science/technology/next-generation-sequencing/sequencing-technology.html>

<sup>4</sup><https://www.pacb.com/applications/whole-genome-sequencing/>

typical run can generate tens of millions of sequence reads, and with a set of experiments that includes biological replicates, control and treatment samples, the total number of reads can reach into the billions [45]. All the reads are later used to map or assemble the complete genome through a matching process, and then it can be analyzed using specific computational tools. All reads and their scores must be stored so the genome can be re-sequenced when better assembling or mapping methods or more accurate references become available.

A typical format for storing this raw data is FASTQ [46], which has become a de facto standard. A FASTQ file represents reads in plain text, as shown in Figure 2-2. Although the representation of genomic information may vary, it generally contains four fields for every read: 1) a record identifier (ID) preceded by '@' with a specific structure for representing experiment meta-data; 2) a read sequence with the actual DNA nucleotide bases {A,C,G,T,N}; 3) a separator or commentary field (commonly discarded by compressors) starting a '+' character and; 4) quality scores (QS) as ASCII characters in a variable range indicating the probability of a sequencing error in each base of the sequence. The exact ID structure, the range of QS, and the length of the reads sequences depend on the sequencing platform used to produce the FASTQ file. Typical variations in different versions of FASTQ files may be related to the identifiers, quality scores alphabet and the use of the identifier field. While for short genome species such as viruses a FASTQ file can be in the order of tens of megabytes, for humans it is in the order of tens of gigabytes.



**Figure 2-2.** Basic structure of the FASTQ format: identifier, read sequence, separator and the quality scores [46].

Recent approaches in compression of genomic data have considered this type of information's characteristics to develop specialized algorithms, obtaining better results than general purpose tools [45] [47] [24] [27] by exploiting traits such as the redundancy of information present in multiple individual genomes of the same species. The attempts on genetic data compression have led to a wide variety of algorithms that can be classified depending on the methodology used to exploit the redundancy of genomic information.

Raw data in the FASTQ file is much more redundant than genomes (which are processed data), being also very different in size and structure. On average, a FASTQ file requires many times more space than the respective genome of the same individual. Since reads are the biggest part of the

storage problem, they were selected as our main compression objective.

### 2.1.2. Trends in Compression of Genomic Data

Data compression seeks to reduce the number of bits used to store or transmit information. It encompasses a wide variety of compression techniques [16], which can vary between software and hardware implementations. The level of specialization of the current compression algorithms is oriented to exploit the specific characteristics of each type of data to be compressed, allowing to optimize its performance, this is why specific compressors for genomic data have been developed in recent years. The effectiveness of a certain compression algorithm can be measured by computing its compression ratio, one of its various definitions [48] is the rate between the size of the input size and the size of the output file, as follows:

$$\text{Compression Ratio} = \frac{\text{Original File Size}}{\text{Compressed File Size}}$$

Although the vast majority of NGS data is currently compressed through general purpose methods, in particular gzip, bzip and its variants, the need for improved compression has led to the development of a number of techniques specifically for this case [29]. This takes us to consider basic characteristics of biological sequences like the reduced alphabet, data distribution and DNA metadata, in order to obtain better compressing ratios and lower execution times. All instances of genomic information share a set of particularities that can be exploited in compression techniques. It is frequent to find repetitions of substrings in a complete chain of DNA (e.g. repeating occurrence of AAACGT). DNA sequences within the same species are highly repetitive. Approximately 0.1% of a genome is unique to an individual, the rest is shared by all members of the species [12]; an issue that must be taken advantage of by compressors.. Achieving higher compression ratios is one of the most important objectives in projects where new algorithms and implementations are created for genomic data [4].

While early compressors for DNA focused on *genome* compression only [26, 49], during the last decade there have been many efforts in developing specialized compressors for different types of DNA data in different file formats. In 2013, the SequenceSqueeze competition focused on promoting specialized compression for FASTQ files due to their relevance [50]. Several lossless compressors for FASTQ have been released ever since [13, 22, 27, 29, 31–38, 50, 51, 51]; currently most of them have been reviewed and tested in detail [16, 24, 52, 53]. Researchers have adopted two trends for compression of DNA strings: Horizontal or non-referential, and vertical or referential. Recent instances of both kinds of algorithms are presented as follows. The main difference between both approaches is the dependency of referential methods on an external appropriate pattern (the reference) to perform the compression, while non-referential compressors only analyze

the bases within the reads.

Non-referential methods are particularly important because they are typically used to complement referential approaches, improving the compression capabilities of such methods. This will be a fundamental issue in the design of the compressor to be presented in chapter 4.

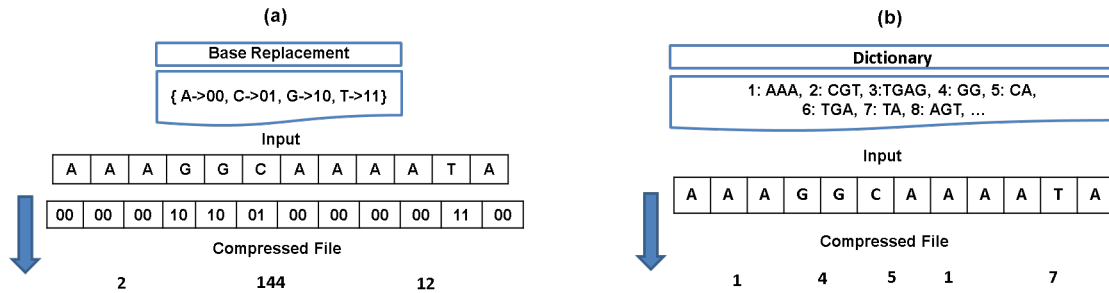
### 2.1.3. Non-Referential Algorithms

The statistical principles that form the basis of traditional compression approaches have led to the development of specific purpose compression algorithms. Non-referential approaches produce self-contained files that can be decompressed with no dependency at all. which is suitable in the absence of an adequate reference or when trying to apply the same method to sequences of different organisms.. According to Wandelt [16], work in this type of algorithms can be further subdivided into four compression strategies :

*Bitwise methods:* According to some authors [16], a straight-forward compression technique for DNA sequence data is the encoding of four bases within one byte via bit encoding (Figure 2-3a). This strategy takes advantage of the short set of symbols present in DNA sequences, and encodes each one in a 2-bit representation (covering the four symbols) instead of the 8-bit traditional representation. [54] applies this principle, achieving a compression ratio of 4. The popular tool DRSC [39] describes a particular compression scheme, employing arithmetic and Huffman encoding, whose results vary between 4.4 and 5.3 compression ratios. Others successful usage of this technique have been reported: combined with dictionaries [55, 56], run-length encoding [16, 57] and others [58].

*Dictionary-based methods:* Dictionary-based methods are compression schemes are generally independent of the specific characteristics of the input data. As shown in Figure 2-3b, the overall strategy is to replace repeated data elements (here: DNA subsequences) of the input with references to a dictionary. Repetitions can be usually detected by bookkeeping previously occurring sequences. This means that the whole dictionary does not need to be stored along with the compressed data in some approaches, since it could be reconstructed at runtime during the decompression process. The procedure used to generate the dictionary and to manage pointers is what varies between approaches. Examples of dictionary-based approaches are Lempel-Ziv-based compression algorithms, such as LZ77 or LZ78 [17]. Current methods for dictionary-compression reach compression rates between 4:1 and 6:1 depending on the frequency of repeats in the genomes being compressed.

Dictionary-based compression improves compression ratios and allows random access to the compressed DNA sequences, but can increase the cost of execution given the multiple iterations needed.



**Figure 2-3.** Basic compression methods: (a)Bitwise, (b)Dictionary based [16]

Also, storing the dictionary (even partially) requires additional resources. However, they can achieve high compression on average, not only for DNA context.

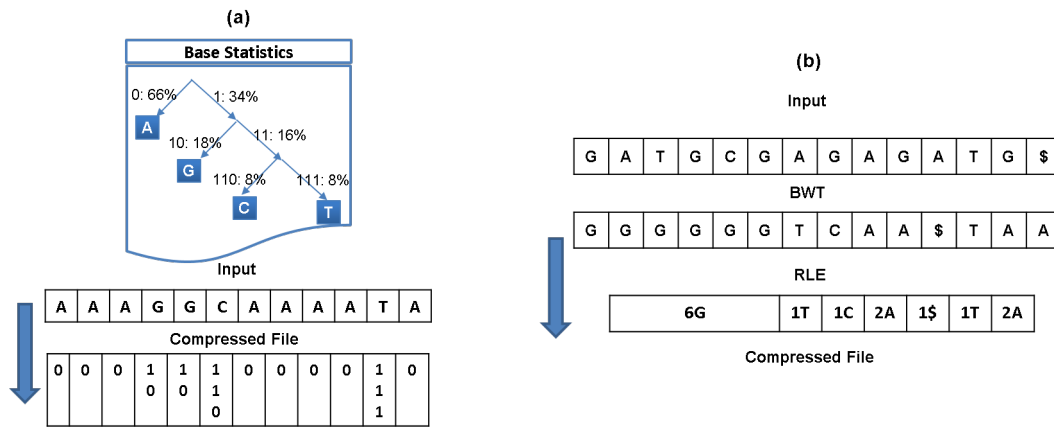
Representative work in this regard is included in [59], which iterates over the whole file multiple times to generate the dictionary, and stops when it has not significantly changed, reaching a compression ratio of 5.5. Kaipa [60] deals with non-uniform distributed DNA sequences by dividing the input in blocks and processing each one separately, using a hash table to detect repeats; this approach shows a compression ratio between 5.3 and 5.7.

*Statistical methods:* Statistical algorithms create a model from the input data, representing it as a probabilistic tree data-structure, where most repetitive sequences are assigned shorter codes (Figure 2-4a). For simplicity, we distinguish two broad classes of codes: fixed codes, such as Golomb codes [61], Elias codes [62] and Fibonacci codes [63], and variable codes, such as Huffman codes [18]. One of the most commonly used and best understood statistical encodings is Huffman encoding [20]. It uses a variable-length code table derived from estimated probabilities for the occurrence of each possible symbol. A binary tree is created in which leaf nodes correspond to symbols and edges are labelled with probabilities and the derived codes. The resulting Huffman code table has to be stored as well. Compression ratio of statistical algorithms are usually between 4:1 and 8:1 [16]. It depends mainly on the existence of detectable patterns in the input and the available memory for construction of frequency distributions. It requires more resources than other approaches, but is useful for long sequences or even for databases, where a single statistical model could be constructed for all the elements stored. From Shannon's entropy coding theory [64], optimal encoding of these data from a compression standpoint depends on their distribution in order to assign shorter binary codes to more probable symbols (integers).

A method that creates various program instances that predict the next symbol in a sequence with different heuristics is presented in [65], this method exhibits a compression ratio of 4.73. [66] proposes the creation of the same program instances but in separate segments of the DNA sequence, prioritizing the heuristics that exhibit a greater compression ratio; by this strategy, a compression ratio between 5.3 and 5.7 is obtained.



*Transformational methods:* The sequence undergoes a preliminary process, where a special transformation is applied (Figure 2-4b). This step commonly enhances the performance and compression ratio of previously described techniques. One example of such transformations is the Burrows-Wheeler transform [28]. Though these methods do not comprise a separate form of non-referential compression, they mark a trend in recent solutions proposed to this problem since the permuted sequence is easier to compress (with other techniques as move-to-front-transform or run-length encoding) than the original input [27, 67, 68].

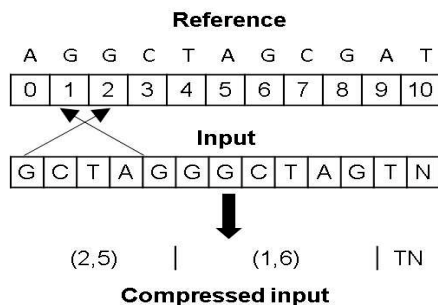


**Figure 2-4.** Basic compression methods: Statistical (a), Transformational (b) [16].

### 2.1.4. Referential Algorithms

DNA sequences from related species exhibit high levels of similarity. This fact is exploited by referential compression schemes, whose key idea is to encode sequences with respect to another reference sequence(s) [16, 40]. Figure 2-5 shows an example of a basic referential scheme.

Algorithms that follow this method have only recently been adopted by biologists with the appearance of research projects that focus on re-sequencing genomes rather than sequencing genomes of new species over the last years. One of the biggest challenges in referential compression algorithms is to efficiently find long matches between the reference and the read sequence to be compressed. Current approaches use heuristics based on index structures, hash-based structures, graphs, suffix trees, alignment data among others [13, 42, 69–71]. Once matches and mismatches are determined, another challenge is to find a space-efficient encoding scheme. A second challenge is to find a space-efficient encoding of the matches and mismatches between the input and the reference, and other data blocks needed. Finally, finding a proper reference sequence can be non-trivial, for which they have been commonly selected using biological criteria. In the recent



**Figure 2-5.** Referential compression. A sub-sequence is represented as the pair  $(x,y)$ , where 'x' is the start position on the reference sequence and 'y' expresses how many symbols of the sub-sequence are represented. Short differences may be encoded as raw strings.

years, building references from the input itself has arisen as an alternative.

A wide range of compression ratios have been reported [16, 53, 72] for genomes reference based compression. If computational resources and a good reference are available, this approach is ideal for the compression of long sequences or collections of sequences, since very high ratios can be achieved (e.g. 400:1 [16]). However, it should be noted that decompression requires exactly the same reference used for compression [22]. This is why a referential approach makes sense as long as one reference is used to compress multiple sequences.

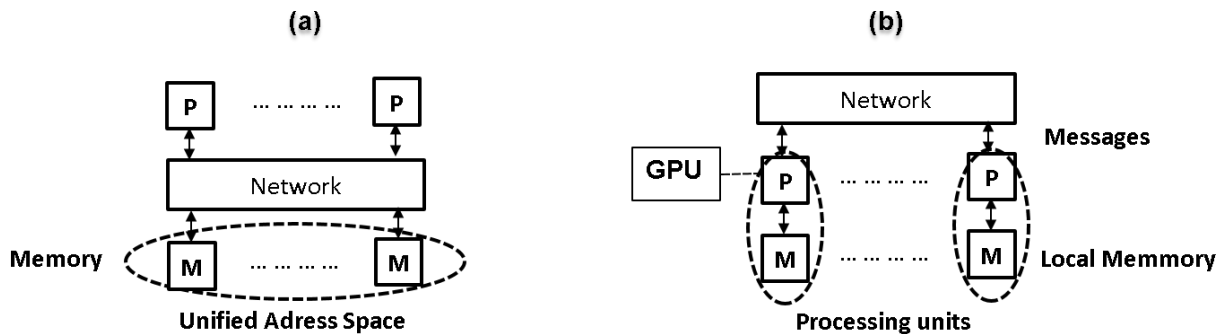
In spite of its potential, by 2014 referential compression had not been applied to NGS raw data compression. Hence there was a clear opportunity to measure the impact of applying a referential approach to compress the DNA stream inside the FASTQ. To get a deeper understanding of this situation and the problem to be tackled, we performed a comprehensive review of the state of the art, which will be presented in the next chapter.

## 2.2. High Performance Computing Systems

Considering that the long execution times of compression algorithms are often a barrier for their adoption, it is important to find ways of accelerating them. High-performance computing (HPC) systems are commonly used to perform bioinformatics analyses and store the data produced. Such HPC systems are complex infrastructures made out of many central processing units (CPUs), memories and arrays of disks for storing files. The availability of this type of parallel architectures allows us to take advantage of parallel algorithms that run faster and can tackle one of the greatest challenges to efficient file storage: compression and decompression runtimes.

In HPC architectures, a node can have multiple processor chips, and a single chip typically in-

cludes multiple processing cores (CPUs). Furthermore, each core is often able to run multiple execution threads simultaneously. Multiple nodes are connected via a network infrastructure, such as 10 or more Gbps Ethernet or Infiniband. There are two main models of HPC architecture, the shared memory parallel (SMP) and distributed memory parallel (DMP) (see Figure 2-6). In a shared memory architecture nodes are individual computers with a memory system that is shared among all CPUs in it. In case of distributed memory each node can only directly access their local memory, and processes running simultaneously pass data to one another in the form of messages through the communication channel [44].



**Figure 2-6.** Multiprocessors Architecture. (a) Shared Memory scheme. (b) Distributed Memory scheme, modified from [73]

The commercial domain of HPC technologies contains mainly four types of on-chip processing architectures: CPUs, which comprise traditional multiple core processor architectures such as Intel Xeon and AMD Opteron; GPUs (Graphics Processing Units), accelerators optimized for handling great amounts of data in parallel; MICs (Many Integrated Cores), a technology by Intel that integrates a large amount of processors (termed manycores) with enhanced vector operation capabilities in a single chip, present in the Xeon family; and FPGAs (Field Programmable Gate Arrays), accelerators encompassing numerous sets of logic gates and tiny processing elements in a single chip, whose connection layout can be altered, generating a reconfigurable hardware solution. Also, paradigms of parallel programming such as single instruction, multiple data (SIMD) have become relevant recently. SIMD and the multi-threaded programming will be discussed in section 2.3.

Although the multicore and manycores architectures have a huge potential to tackle present and future applications, a key issue is still open: how can developers map an application onto such a multicore platform fast and efficiently, while profiting from the potential benefits of parallel processing. Obviously, programming a multicore system requires some sort of parallel programming model for algorithm design, and an appropriate programming language and supporting libraries must be selected. A developer can choose between writing code manually to have detailed control on the parallel execution flow, or use automatic or semi-automatic libraries to be relieved from

the tedious and error-prone manual parallelization process. Though the quality of automatic parallelization has improved in the recent years, fully automatic parallelization of sequential programs by compilers remains as a challenge, due to the need for complex program analyses and the unknown factors (such as input data range) during compilation [73].

A number of parallel programming languages (OCCAM, HPF, OpenCL, CUDA), libraries (PThreads, MPI, OpenMP), and other software solutions ( auto-parallelizing compilers, e.g. SUIF, Paraphrase-II, Paradigm, Compaan) have been developed in the last decades, yet the main problem in most cases has been either their low acceptance among developers or the difficult to achieve optimal performance in some cases.

By the time of our state of the art review, all the most relevant FASTQ compressors [22, 29, 50, 51, 69, 70, 74, 75] (evaluated later in this thesis) were implemented for CPUs processors. In order to ensure maximum compatibility with the hardware that is commonly available in bioinformatics research centers, in 6, we will present the efforts made to accelerate of the compression solution developed in this research though CPU's parallelism.

## 2.3. Parallel Programming

The main idea of parallel programming is harnessing the processors architecture executing simultaneously parts of the algorithms, which decreases the execution time required. Algorithms and multiprocessing architectures are closely tied together. We cannot speak about parallel architectures without speaking of the parallel algorithms to harness it [76]. Flynn's taxonomy [77] define different types of parallelism in computer architectures, according to the number of concurrent instruction and data streams available. In this section we present the two types of parallelism we considered in this research (SIMD and multi-threaded) as well as the programming models used.

### 2.3.1. SIMD Level Parallelism

Data-level parallelism (DLP) refers to a type of parallel processing in which the same computation is applied simultaneously to several data elements. One common instance of DLP is single the single instruction multiple data (SIMD) paradigm. SIMD parallelism is implemented through instruction-set extensions available in almost any modern processor. Such instructions must be explicitly invoked by the programmer through intrinsics or directly in assembly language.

SIMD instructions have been successfully used in applications characterized by: (a) inherent DLP, (b) typical small data types (8, 16 and 32 bits), (c) recurring memory access patterns, (d) localized recurring operations performed on the data and (e) data-independent control flow [78]. In spite of its benefits, SIMD brings important difficulties when parallelizing some algorithms due to the following reasons: SIMD may not be suitable for all algorithms (or not fore the entire code), SIMD

programming requires human work (automated parallelism is not available) which involves handling numerous low-level details, and finally the instruction sets used are architecture-dependent so binary compatibility is lower.

Although former SIMD ISA extensions were simple (e.g. MAX-1, AltiVec and SSE), successive generations have become more sophisticated offering wider SIMD registers to process more elements per instruction and using more complex instructions to operate on them. The first extensions were implemented by manufacturers using 64-bit registers, and since then they have evolved scaling SIMD extensions to 128-bit, 256-bit and 512-bit registers [78]. The inclusion of SIMD extensions in general purpose processors is a very common trend nowadays, being present in most modern architectures (Intel, AMD, ARM). It is expected that the widths and capabilities of SIMD support will improve significantly in future microprocessor generations [79]. Some authors predict that the SIMD support found in commodity microprocessors will eventually resemble the instruction sets of classic vector architectures traditionally found previously in supercomputers [80].

In this thesis, the SIMD optimizations were implemented through the usage of Intel's intrinsics, which correspond to C style functions that provide access to Intel instructions (including Intel SSE, AVX, AVX-512), without the need to write assembly code. For a deeper understanding of the instructions discussed in this chapter, we invite the reader to consult Intel intrinsics guide at their website<sup>5</sup>.

In chapter 6, we will focus mainly on exploiting DLP (through SIMD instructions) to accelerate compressor. Also we will present some efforts to explore the possibilities of getting an additional performance gain combining data-level and threads level parallelism in the aforementioned task.

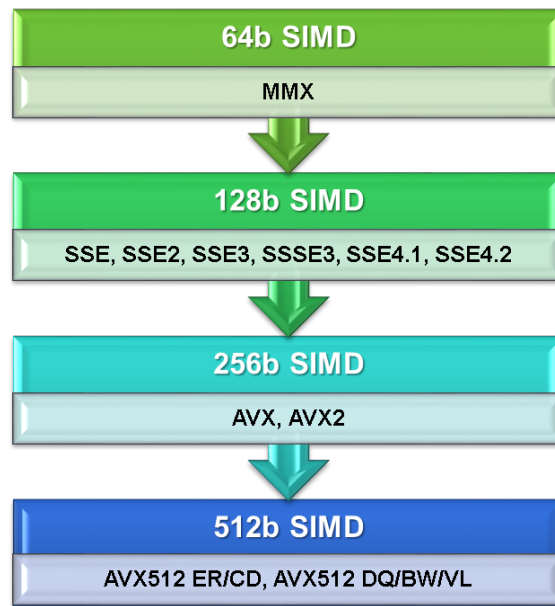
### 2.3.1.1. Intel™ AVX-512

Intel™ Advanced Vector Extensions 512 [81] (Intel™ AVX-512) is a set of new advanced instructions that can accelerate performance for demanding computational workloads, including useful instructions such as gather/scatter as well as masked operations, conflict detection instructions, non-trivial SIMD instructions, support for many data types and vector length extensions.

The SIMD capabilities in x86-based microprocessors have moved from simple 64-bit multimedia extensions in MMX to 128 bits in SSE, and to registers of 256 bits in AVX. Intel's AVX-512 increased the width of the registers to 512 bits, with plans to expand to 1024 bits in future generations [82] ( See Figure 2-7).

AVX-512 are 512-bit extensions to the 256-bit Advanced Vector Extensions SIMD instructions for x86 instruction set architecture (ISA) proposed by Intel in July 2013, and supported in Intel's Xeon

<sup>5</sup> <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>



**Figure 2-7.** Evolution of Intel’s SIMD extension on top of x86/x87.

Phi x200 (Knights Landing) and Skylake-X CPUs; this includes the Core-X series, as well as the new Xeon Scalable Processor Family and Xeon D-2100 Embedded Series. Compared to the Intel Advanced Vector Extensions 2 (Intel AVX2) instruction set, Intel AVX-512 doubles the number of vector registers available, and each vector register can pack twice the number of floating point or double-precision numbers.

### 2.3.2. Thread Level Parallelism

A multicore system usually refers to a multiprocessor system that has all its processors on the same chip [76]. This close packing allows a fast interprocessor communication without too much power consumption. In order to harness the computational power of modern multi-core CPUs, a program must use multi-threading programming. Such technique, however, will rely on the existence of enough coarse grain parallelism in the involved algorithms to exploit the large number of cores. In practice, two multi-core frameworks are commonly used for multi-thread programming: a low-level approach using POSIX threads library (Pthreads) and a higher-level approach using the OpenMP interface. Both of them have been widely applied in bioinformatics to provide the computing performance required by this workload [44, 78].

In this work we use OpenMP, a programming interface for shared memory parallel computing. It offers a higher level of abstraction compared to the PThreads library. The main advantage of the OpenMP is simplicity compared to other frameworks, which is relevant since the difficulty for extracting parallelism from traditional sequential code is one of the main factors that prevent

the performance increase in multi-threaded programming. It is available on various environments (including Unix and Windows) and programming languages (C, C++, FORTRAN).

## 2.4. Summary

The amount and importance of genomic data will continue to increase in the near future, hence the need to create strategies to efficiently compress it, for its later processing, storage or transmission. In this chapter, we presented the basics of genomic data compression and a brief review of the most relevant tools in the state of the art. Genomic data consists of strings of characters representing base-pairs. The bulk of the information stored in genomic databases is comprised by short such sequences called reads, a direct product of genome sequencing.

Bioinformaticians have developed a variety of algorithms to compress this type of information, mainly taking one of two approaches: referential and non-referential. Referential algorithms exhibit lower execution times at higher compression ratios, although they require a read alignment stage in order to achieve an optimal representation of the read in the reference. Non-referential algorithms encode the compressed file using only information contained within the same file, i.e. additional information is not required to compress and decompress the DNA sequence. On the other hand, referential algorithms compress DNA sequences by creating references to other similar DNA strings, which may be associated to biologically close individuals.

Due to the huge size of genomic data files, which requires lots of computing resources for its processing, it has been necessary to maximize the use of available hardware. This has led to applying strategies of parallel programming in order to reduce the time required by the compression algorithms.

In the following chapters we will apply the fundamental ideas explained here in order to develop our compression method.

### 3. Evaluation of the State of the Art in Genomic Data Compression

Referential and non referential compressors for genomic data have been developed during more than a decade. Most of the proposed software has been designed to exhibit a better performance under particular conditions. At the beginning of this research, existing reviews of the state of the art in FASTQ compression were still incipient, being mostly theoretical, or leaving uncovered the analysis of important metrics; which resulted in lack of clarity about the real top performers in that extent.

In our first review of the literature, we found highlighted researches about non-referential compression of FASTQ files, reporting different metrics, capabilities and restrictions for each proposal. At the same time, we did not find experiences related to the referential compression of such files. In the absence of an objective experimental comparison, we decided to perform a comprehensive evaluation of the current state of the art of non-referential compressors, using local data to compare the different proposals presented to date.

In this chapter, we present the methodology and results of a performance evaluation of non-referential NGS data compressors. We performed a series of benchmarks over diverse specialized genomic compressor tools, and general-purpose compression tools such as PBZIP, P7ZIP and PIGZ; using local genomic input data and considering metrics related to compression ratio, execution speed, parallel scalability and memory consumption. Also, we discuss the main findings resulting from those experiments, as well as the research opportunities observed in the process.

Finally, we present a separate review of the current state of the art in referential compressors for genomic data. This review is essentially theoretical, since no referential compressors for FASTQ had been proposed at the time of our performance evaluation.



### 3.1. State of the Art and Performance Evaluation of DNA compressors

Compression algorithms for DNA have existed for over two decades. Early reviews of the state-of-the-art focused on the theoretical analysis of the algorithmic foundations of compression programs. Most of them were restricted to tools and techniques for compressing the whole assembled genome and other sub-strings of interest, without addressing reads, quality values nor large files such as FASTQ. The first experimental comparisons focused on genome compression [26,49], with exclusive emphasis on the compression ratio.

Most authors who had proposed compression programs for DNA sequences included brief comparisons of performance in order to show their advantages. In 2011, Deorowicz presented his DSRC [39] tool with a concise comparison of sequential performance which spanned the compression rate and speed. That same year, Bhola introduced his non-reference prototype system [83] evaluating compression ratio only; and Yanovsky compared the results of the Recoil program [41] also considering compression speed. All these three authors made their evaluations considering at most, one domain-specific tool besides the one being proposed, bringing more attention to general-purpose compressors. This trend continued in 2012 when Delimitate program [84] was presented and evaluated only against general-purpose tools. That year, KungFQ authors [36] also included a few more domain-specific tools in their experiments. All these reports were limited to sequential evaluations though.

The first extensive survey on the most relevant trends in compression of genomic sequences was presented by Wandelt [16] in 2013. He highlighted the main techniques for genome compression and brought special attention to reads compression. Moreover, he proposed important metrics for the evaluation of such programs: compression rate, (de)compression time, and maximum main memory usage. His work did not include an experimental evaluation due to difficulties to compare and evaluate compression schemes. FASTQ files compression became more relevant, being analyzed by Deorowicz in [4]. He also stated difficulties to compare experimentally the existing tools, hence he did not perform it. In 2013 Bonfield performed an experimental comparison of his algorithm against other tools that also participated in the SequenceSqueeze competition [50], exclusively oriented to FASTQ files compression. His analysis was centered on the contest conditions; evaluating compression ratios, execution speed, and memory required for compression; but he did not consider the parallel scalability of tools. More recently in 2014, Rogusky [51] presented DSRC 2 accompanied by a brief report about its scalability, and Giancarlo published a comprehensive and theoretical review of the state of the art [52].

In general, review works had focused on evaluating sequential performance [85, 86] with a small set of performance variables, and comparing mostly to general-purpose tools. Furthermore, even though general-purpose tools had evolved to parallel implementations and multi-core processors

were already common, the few technical surveys presented to date have not considered the scalability of programs running on parallel high-performance systems. Although a handful of authors had reviewed the state-of-the-art in DNA sequence compression [12, 49, 87], very few articles had evaluated the compression of reads and these are mostly focused on theoretical aspects of the algorithms used [4, 16, 52]. The ones that attempted to do experimental performance evaluation were mostly limited to general-purpose tools and to observing one or two performance metrics in a sequential execution scenario. At the beginning of this thesis, we could not find any article that evaluated traditional performance metrics along with the parallel scalability of the tools that supported parallel execution. This is a very important issue considering the intensive computation required for FASTQ compression, and the common availability of high-performance architectures in bioinformatics centers.

Considering those facts, we performed a detailed technical evaluation of the state of the art of FASTQ compression. The main goals of this evaluation were to:

- Complement the available knowledge with respect to the efficiency of non-referential FASTQ compressors, comparing the general-purpose and domain-specific programs publicly available; evaluating multiple performance metrics in a parallel execution scenario.
- Identify, from an independent analysis, the real capabilities and needs of the state of the art in compression of FASTQ files.

### 3.1.1. Evaluated Compressors

Since general-purpose tools were still widely used in bioinformatics, we decided to evaluate some of them alongside domain-specific programs. After studying dozens of FASTQ compressors proposed [13, 22, 27, 29, 33–38, 50, 51, 88–92], we selected the seven most relevant for this evaluation; for the sake of clarity and because of their merit. In this sub-section we briefly present the theoretical bases of the selected methods, classified by their compression approach).

By the time we performed this review of the state of the art, we only found one work focused on referential compression of read sequences data and it was not designed to work over FASTQ files. In consequence, the following experiments only compare non-referential algorithms. Detailed configuration for each tool is presented in Table **3-1**.

#### 3.1.1.1. General-purpose Compression Tools

We reviewed the literature related to plain-text compression searching for open source tools that have proven to be effective, capable of handling large files, allow multi-threaded configuration

**Table 3-1.** Tools configuration used during experiments.

Program	Configuration options
<i>P7ZIP</i>	<i>default</i> – <i>mmt</i> = [1..20]
<i>P7ZIPbest</i>	– <i>mmt</i> = 1 – <i>mx</i> = 9
<i>PIGZ</i>	<i>default</i> – <i>p</i> [1,2,4,...,20] <sup>a</sup>
<i>PIGZbest</i>	–9 – <i>p</i> 1
<i>PBZIP2</i>	<i>default</i> – <i>p</i> [1,2,4,...,20]
<i>PBZIP2best</i>	–9 – <i>p</i> 1
<i>QUIP</i>	<i>default</i>
<i>FASTQZ</i> – <i>fast</i>	<i>e</i>
<i>FASTQZ</i> – <i>best</i>	<i>c</i>
<i>DSRC</i> – <i>fast</i>	– <i>d</i> 0 – <i>q</i> 0 – <i>b</i> 8 – <i>t</i> [1,2,4,...,20]
<i>DSRC</i> – <i>best</i>	– <i>d</i> 3 – <i>q</i> 0 – <i>b</i> 256 – <i>t</i> [1,2,4,...,20]
<i>SCALCE</i> – <i>gz</i>	– – <i>compressiongz</i> – <i>T</i> [1..2] <sup>b</sup>
<i>SCALCE</i> – <i>bz</i>	– – <i>compressionbz</i> – <i>T</i> 1

<sup>a</sup> In decompression, we only tested it using 1 and 2 threads.

<sup>b</sup> When using 2 threads or more, we configured:–*compression pigz*.

and produce common file formats. Bzip2<sup>1</sup>, 7zip<sup>2</sup> and gzip<sup>3</sup> were selected, being prominent open source tools widely used in bioinformatics [4, 27, 29, 42, 45, 50, 55, 93, 94] (see Table 3-1). In text compression, 7zip and bzip2 have been reported to achieve the highest compression ratios while gzip is recognized as one of the fastest programs [95–97]. We used the respective parallel implementations of bzip2 (PBZIP2<sup>4</sup>), 7zip (P7ZIP<sup>5</sup>) and gzip (PIGZ<sup>6</sup>).

## PBZIP2

PBZIP2 uses pthreads<sup>7</sup> for parallel execution and authors claim it increases its speed almost linearly when running in shared memory architectures [92]. It is based on the Burrows-Wheeler transform [28] combined with Huffman coding compression [20]. This means that the symbols within the sequence are relocated to increase the repetition of certain sub-chains, which are represented more efficiently according to their probability of occurrence. The BWT transformation improves the compression with a simple reversible method, which is convenient for problems with reduced alphabets but huge amounts of data. We used version 1.0.6.

<sup>1</sup><http://www.bzip.org/>

<sup>2</sup><http://www.lzop.org/>

<sup>3</sup><http://www.gzip.org/>

<sup>4</sup><http://compression.ca/pbzip2/>

<sup>5</sup><http://p7zip.sourceforge.net/>

<sup>6</sup><http://zlib.net/pigz/>

<sup>7</sup><http://pthreads.org/>

## **PIGZ**

PIGZ uses the zlib<sup>8</sup> and pthreads libraries. It combines Huffman coding and LZ77 [17] for building a compression dictionary that is optimized according to words repetition. Its decompression algorithm is not parallelized due to its deflate format that must be decompressed serially; the only parallelism strategy is to create additional threads for reading, writing and calculating CRC (checksum) [91]. Hence, during decompression experiments threads were scaled only up to two threads. Version 2.3.1 was used.

## **P7ZIP**

P7ZIP tool is an open source implementation of 7zip developed by Igor Pavlov. 7zip has been reported to achieve a compression ratio up to 40% higher than its competitors. It generates 7z format files using Lempel-Ziv-Markov algorithm (LZMA y LZMA2) [98]. LZMA [99] uses an optimized version of LZ77 with large dictionary sizes and special support for repeatedly used match distances, whose output is encoded using a complex model to make a probability prediction of each bit. Furthermore, compression is improved over LZ77 using a longer history buffer, optimal parsing, shorter codes for recently repeated matches, literal exclusion after matches, sophisticated dictionary data structures, and dynamic programming to select an optimal arithmetic coding scheme . LZMA2 works a simple container format that can include both uncompressed data and LZMA data, possibly with multiple different LZMA encoding parameters. LZMA2 supports arbitrarily scalable multi-threaded compression and decompression. We used version 15.14.

### **3.1.1.2. Domain-specific Compression Tools**

Domain-specific lossless compression tools have been developed according to reference and non-reference approaches [12]. As we will explain below, most of them have been developed hybrid approaches, combining different compression techniques to threat the read sequences.

In this extent, several tools have taken advantage of specific features of the FASTQ file stream, we selected four lossless non-referential compressors for FASTQ format, considering their scientific impact, performance previously reported and support for compressing all the strings included in FASTQ files. Several tools ( [13, 27, 31–38]) could not be further tested due to problems such as: incorrect output results, input data restrictions (file size, read size, fixed read length), runtime errors, excessive runtime or very low compression ratios in preliminary evaluations.

Only the tools described below matched our requirements, configured as presented in Table 3-1. Since the standard is compressing separately the three FASTQ file streams, that is how we present

---

<sup>8</sup><http://www.zlib.net/>

them. Latest stable versions available were used in all cases.

## **FASTQZ**

FASTQZ [50] is a tool written in C++ and runs in two modes: compression (parameter *c*) which is slower but more efficient and encoding (*e*) which runs faster. Each FASTQ file is broken into three separated streams, and then the public domain libzpaq compression library is used for creating context models in ZPAQ format<sup>9</sup>. ZPAQ uses a context mixing algorithm based on PAQ data compression archivers [100] in which the bit-wise predictions of multiple independent context models are adaptively combined. ZPAQ is a comprehensive tool that may use LZ77, context models, BWT, or combinations of them for the compression.

FASTQZ has some input format restrictions, so we had to pre-process our Illumina FASTQ files before tests. We used version 15.

*Identifiers:* IDs are compressed considering the differences between consecutive lines, which are encoded as a numeric field increment in the range 0-255, a match length, and trailing differences. If running in best mode, the names previously encoded are modelled using a mix of four context models, each consisting of: a hash of the column number, the current byte in the previous line, the previous bits in the current byte, and the last 1, 2, 3, or 4 bytes in the current line.

*Quality scores:* The method is tailored to QSs according to the common Sanger variant. Byte codes are used to indicate runs of score 38 up to length 55, or groups of three scores in the range 35-38, or pairs of scores in the range 31-38, or single bytes for other scores; the rest of the scores are 2 and are omitted. In fast mode, no additional encoding is done. In best mode, the resulting codes are modelled using a mix of three direct context models.

*Sequence encoding:* FASTQZ packs multiple bases together, assigning A=1, T=2, C=3 and G=4. Letter N does not need to be coded because it has a QS of 0 that is inserted during decoding. It packs 3 or 4 bases together, whichever numerical packed value does not exceed 255. Finally, any sequence starting with G, CG, or CCG is coded in 3 bytes, and any other sequence in 4 bytes. In best mode, the encoded sequence is compressed using a mix of 6 models ranging from order 0 through order 5 bytes. According to the authors, total memory usage is about 1.4 GB for any file size.

## **QUIP**

QUIP [22] is a tool implemented in ANSI C99 and is based on statistical compression with high order modeling and arithmetic coding; a refinement of Huffman coding. This is very convenient since

---

<sup>9</sup><http://mattmahoney.net/dc/zpaq.html>

it allows a complete separation between statistical modeling and encoding, hence, in QUIP the same arithmetic coder is used to encode each of the streams in the file, but using different statistical models. Furthermore, QUIP uses adaptive modelling in order to achieve a higher compression. QUIP also offers assembly-based compression (option -a), which has been tested before [22,50,85] showing slight improvement in compression ratio while increasing execution times over twice. In consequence, that option is not explored here. We used version 1.1.8.

*Identifiers:* To remove redundancy in IDs, QUIP uses a form of delta encoding. IDs are split into separate fields which in order to compare them consecutively. Tokens that remain the same from read to read are compressed using arithmetic coding. Numerical tokens are stored directly or as an offset from the token in the same position in previous read. Otherwise, non-identical tokens are encoded by matching as much of the prefix as possible to the previous read token before directly encoding the non-matching suffix.

*Quality scores:* An order-3 Markov chain is used to model QSs in correlated positions. Additionally, QUIP bins QSs in adjacent positions to reduce control parameters and optimize the model accuracy. Reads with highly variable QSs are encoded using separate models.

*Sequence encoding:* To compress sequences of bases, they adopt a simple model based on high-order Markov chains. The base at a given position in a read is predicted using the preceding 12 positions. Authors claim that it uses a significant amount of memory but requires very little computation. Though less efficient at compressing short files, after compressing million of reads, the parameters are tightly fit to the base composition of the dataset in order to highly compress the remaining reads. In consequence, compressing larger files leads to a tighter fit and higher compression.

## DSRC

DSRC [51] is a multi-threaded tool written in C using the Boost libraries<sup>10</sup>. It offers best (-d3) and fast (-d0) execution modes, which basically differ in applying or not LZ-matches encoding on the sequence stream. In DSRC, I/O operations are executed using a single thread while several threads perform the compression or decompression tasks [39]. DSRC has important extra features, allowing fast random access to the records of a compressed file and offering options to perform lossy compression on QSs (-q[1,2]). We used version 2.0.

*Identifiers:* DSRC treats them as a concatenation of several fields with separators. Since the elements in these lines can be very heterogeneous, statistics are gathered so that each element can be compressed in an efficient way. The techniques used include: compact encoding of constant fields, recognition of numeric and non-numeric fields, efficient encoding of columns with fixed characters

---

<sup>10</sup><http://www.boost.org/>

in non-numeric fields, detection of the numeric fields amenable to differential coding and entropy coding.

*Quality scores:* DSRC dynamically decides which of the two strategies available is better suited for storing Qs. The first approach uses an order-1 Huffman coder with the context related to the position in the read, or an order-1 Huffman coder of the run-length-encoded quality stream. In the second method, the quality values are compressed arithmetically with context lengths up to 6.

*Sequence encoding:* Symbols can be encoded in three ways, which is autonomously decided by the tool. In the first one, each base is stored in 2 bits. In the second one, a Huffman coder is applied on the symbols. The last method uses an arithmetic coder [101] combined with contextual probability estimation of orders up to 9.

## SCALCE

SCALCE [29] is mainly a C++ compression booster based on the Locally Consistent Parsing (LCP) technique, which provides an efficient way of reordering reads to improve the compression ratio and the compression runtime, independently of the compression algorithm used. This approach has shown good performance and high scientific impact. It works better when combined with external compression tools, so we tested the options: SCALCE+gzip (or pigz when several threads are configured) and SCALCE+bzip2. Although SCALCE has been evolving since the original version published in 2012, subsequent changes have not been widely documented. We used version 2.8.

*Identifiers:* Lempel ziv strategies are used to compress IDs [52]. No additional details are provided by authors.

*Quality scores:* SCALCE uses arithmetic coding to compress Qs [52]. To reduce runtime, it calculates the frequency table for the alphabet of Qs only from a reasonable subset of them (1 million Qs). Lossy compression of Qs is an option, but it was not tested since we are only interested in lossless compression. No additional details are provided by authors.

*Sequence reordering and encoding* The purpose of reordering reads is to group highly related reads, in fact those reads that ideally come from the same region and have large overlaps; boosting gzip and other Lempel Ziv 77 based compression methods. This is achieved by observing sufficiently long core sub-strings that are shared between the reads, and clustering such reads to be compressed together. This reorganization acts as a fast substitute for mapping-based reordering. The core substrings of the boosting method are derived from the LCP method [102]. LCP is a combinatorial pattern matching technique that aims to identify building blocks of strings. For each read, LCP simply identifies the longest(s) core(s) substring(s). The reads are "bucketed" based on

such representative core strings and within the bucket, ordered lexicographically with respect to the position of the representative core. Reads in each bucket are then compressed using Lempel-Ziv variants or any other related method.

## 3.1.2. Experimental Methodology

### 3.1.2.1. Datasets

Input data were provided by experts from the National Genome Sequencing Center at the University of Antioquia, Colombia. They selected representative data from their daily work, in this case, containing transcriptomes in three FASTQ files obtained with Illumina HiSeq <sup>®</sup>technology. A transcriptome is defined as the set of messenger RNA resulting from the translation of the genome under certain conditions [103]. The structure of this type of sequence is similar to DNA raw data, combining five symbols: A, G, C, T, N [104]. Also, the computational load represented by the dataset was also equivalent to that of genomic data. Qs were in Phred+33 coding, which is used in the Sanger and Illumina format from version 1.8 on; the most used technologies nowadays. File sizes were between 7.7 and 8.1 G.

### 3.1.2.2. Experimental Design

Each of the selected programs were used to compress and decompress the files in the dataset. Although care was taken as to run the tests when the server was free, three replicates of each test were made in order to filter out possible additional "noise" affecting the time measures. The first set of tests was made with sequential configurations (single-threaded). Then, for the tools that allowed it, the number of threads was scaled up until 20 threads. Metrics recorded were: compression ratio, processing speed (as cumulative throughput expressed in Megabytes per second - MBps), parallel scalability and memory consumption.

Tests were performed on a server with four Intel Xeon E5-2620 (6 cores each, with 2-way hyper-threading), 2.00 GHz, 15MB cache, 96 GB RAM (shared memory architecture) and 1.1 TB SATA disk with Centos 6.5 operating system (64 bits).

Time was obtained using the unix *time* command, and normalized to the file size to calculate throughput. Memory was observed using Valgrind software<sup>11</sup>. All reported values are means, except for cumulative memory measurements which are maximum. The percentage of variance in results was always less than 10 % and thus it was not necessary to discriminate results according to the different test files.

---

<sup>11</sup><http://valgrind.org/>



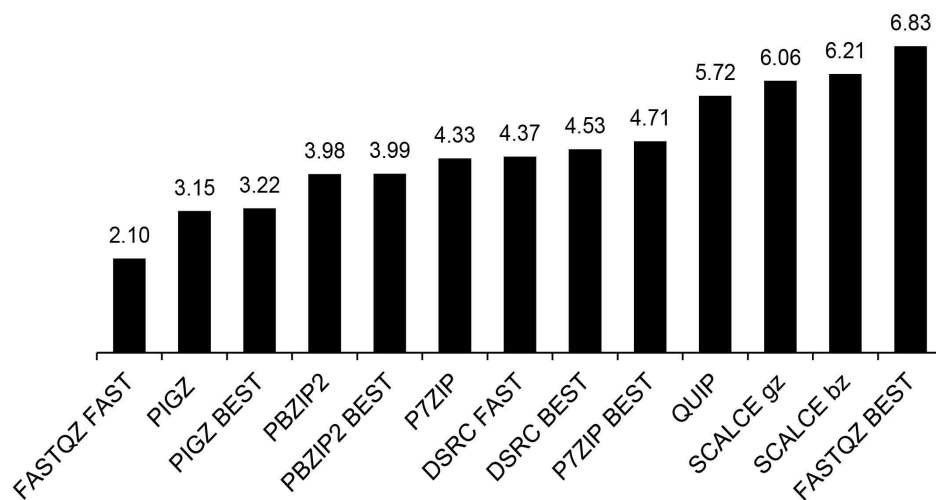
### 3.1.3. Results and Discussion

We first present compression ratio results and then divide the processing speed results in two: sequential and multi-threading performance, where we also present the parallel scalability. Last, we analyze memory consumption.

#### 3.1.3.1. Compression Ratio

In order to explore the trade-off between compression ratio and speed BZIP2, 7ZIP, PIGZ, FASTQZ, SCALCE and DSRC were run in both configuration modes: *fast* (default configuration) and *best* (to achieve a higher compression ratio).

Figure 3-1 shows compression ratios obtained from all the evaluated tools. FASTQZ-best achieved the best compression ratio of all, being almost 70% better than the best general-purpose tool (P7ZIP). SCALCE-bz achieved the second best compression. On the opposite side, FASTQZ-fast achieved the lowest results; next were PIGZ and PBZIP2 in both configurations (default and best). Best modes of execution provided a highest compression ratio; however, in all cases this improvement tended to be insignificant. With two exceptions, all domain-specific tools performed better than all general purpose tools: FASTQZ-fast performed worst than any other tool, and DSRC (both configurations) performed slightly worst than P7ZIP.

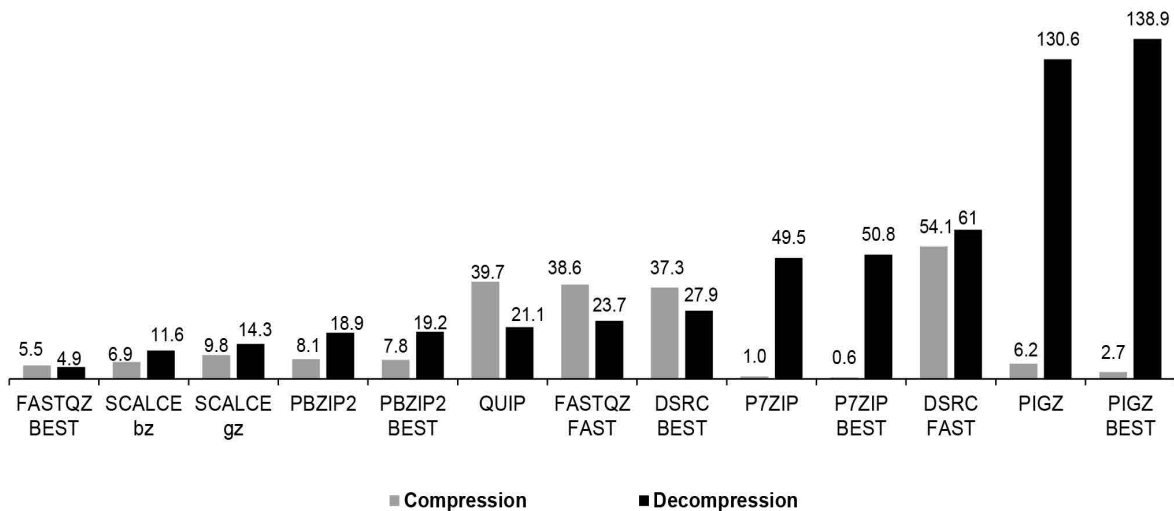


**Figure 3-1.** Compression ratio of evaluated programs, represented by the original file size divided by compressed file size.

### 3.1.3.2. Single-thread Performance

The single-thread evaluation intended to compare tools that do not run in parallel. However, an exception to this was FASTQZ. In such a tool (both modes) the number of threads cannot be set by the user, instead, the tool dynamically activates a multi-threading mode by default. We monitored its execution and noticed that 2-4 threads (at most) were used dynamically. Since the multi-threading results aimed mostly at analyzing scalability, which is not possible for FASTQZ, we decided to include its results in Figure 3-2. In any case, running times were so high for FASTQZ that this does not affect the interpretation of results.

During compression (Figure 3-2), DSRC-fast clearly outperformed all the other tools, reaching 54.14 MBps. QUIP came second with 39.74 MBps. Among general purpose tools PBZIP2 was the fastest compressing 8.1 MBps, followed by PIGZ (6.2 MBps). P7ZIP was the slower compression tool processing only 1.0 MBps. General-purpose compressors were always slower than the any domain-specific tool, except FASTQZ-best and SCALCE-bz.



**Figure 3-2.** Single-thread throughput (MBps) during compression and decompression. Ordered by decompression speed

In decompression, PIGZ (both modes) outperformed the rest of evaluated tools decompressing over a 130 MBps rate, more than twice faster than the second faster tool. DSRC-fast throughput was also noticeable, achieving 61.07 MBps. Both P7ZIP configurations were the third faster decompressors. Quip decompressed 40 MBps less than DSRC-fast and slightly faster than PBZIP2; while FASTQZ-best and SCALCE (both modes) had the worst performance. Also, it was noticeable that even when DSRC-best and FASTQZ-fast runtime were comparable, DSRC-best achieved a compression ratio twice higher.

The effect of the *best* compression configuration (highest compression ratio) is summarized in Table 3-2, it was calculated in comparison to fast (default configuration) results. For compression ratio the number in the column shows how much the compression ratio improved in *best* mode, while for compression and decompression speed it shows how much slower it was. In general, it makes no sense of investing more runtime in the evaluated methods using the *best* configuration, to get a non-significant improvement in the compression ratio.

**Table 3-2.** Effect of *best* configuration (highest compression ratio) in performance. We used = when the difference was 5% or less, and  $\approx$  when it was 10 % or less.

Program	Comp. Ratio	Comp. slowdown	Dec. slowdown
<i>P7ZIP</i>	$\approx$	$\sim 1.7\times$	=
<i>PIGZ</i>	=	$\sim 2.3\times$	$\approx$
<i>PBZIP2</i>	=	=	=
<i>FASTQZ</i>	$\sim 3.5\times$	$\sim 7\times$	$\sim 4.5\times$
<i>DSRC</i>	=	$\sim 1.5\times$	$\sim 2.2\times$
<i>SCALCE</i> *	=	$\sim 1.5\times$	$\sim 1.23\times$

\* For SCALCE we assumed the bz configuration as the *best* one.

For all general purpose tools, *best* configuration did not affect decompression speed, this may happen if the dictionary is stored along with the compressed data (does not need to be calculated) or if the dictionary is static, with the same size independently of using the **best** or *fast* method. It is interesting to notice that for all the tools (except FASTQ) *fast* and *best* modes achieved comparable compression ratios, while having differences in compression runtime. For P7ZIP this difference is particularly important, considering that the default version took at least two hours to compress any of the files. For domain-specific tools, this difference was also significant during decompression.

### 3.1.3.3. Multi-threaded Performance

Single-threaded experiments stood clear that in all cases this *best* configuration did not improve significantly the compression ratio; but it had a negative impact in the runtime. That also happened with the SCALCE-bz configuration, being slower but not better than SCALCE-gz. Therefore, the only *best* configuration included in the multi-threaded experiments was the DSRC-best, which appeared to scale well in our preliminary tests.

For these experiments, the level of parallelism (number of threads) was set to vary from 1 to 20 threads; only DSRC (both modes), P7ZIP, SCALCE, PBZIP2 and PIGZ allowed this configuration. Compared to sequential executions, differences in compression ratios were negligible.

Cumulative throughput during compression is shown in Figure 3-3a. All tools, increased throughput as the number of configured threads increased, achieving their maximum performance when

using the maximum of 20 threads. The exception was SCALCE, it achieved the maximum performance (almost 10 MBps) using 12 threads. DSRC-fast was clearly superior reaching more than 670 MBps. DSRC-best maximum throughput was 350 MBps, PBZIP2 reached 100 MBps, PIGZ over 80 MBps and P7ZIP compressed at a maximum of 8.2 MBps.

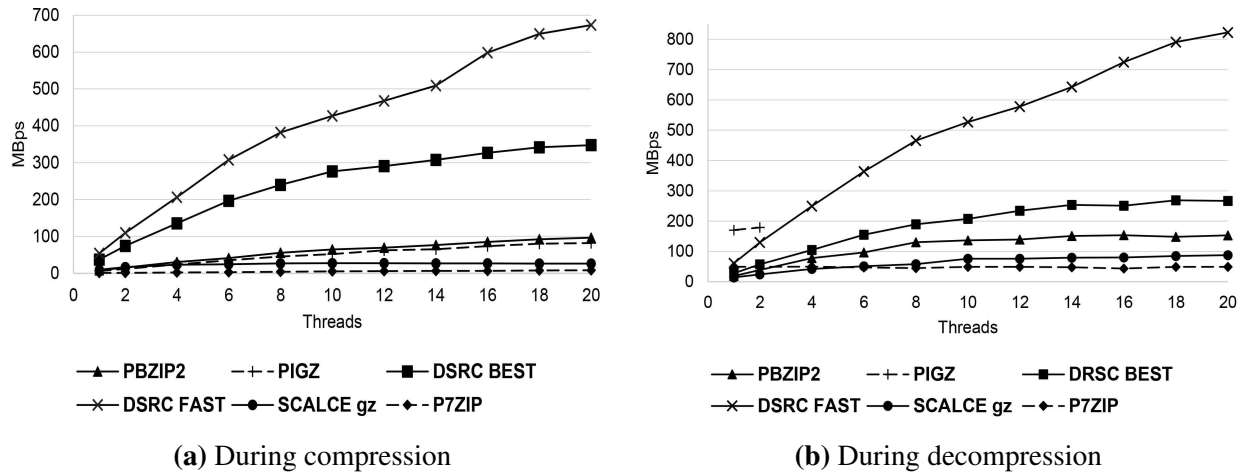


Figure 3-3. Multi-threaded throughput

Decompression throughput can be observed in Figure 3-3b. Best performance was reached by DSRC-fast which got to decompress up to 822 MBps using 20 threads, more than  $4.5\times$  faster than the best general-purpose tool. DSRC-best decompressed up to 268 MBps using 18 threads. It was faster than PIGZ, but required much more computing resources since PIGZ only used two threads to achieve 179 MBps. PBZIP2 maximum throughput was over 150 MBps using 16 threads, being  $1.75\times$  faster than SCALCE gz using 20 threads. P7ZIP had the worst performance of all, showing no improvement when increasing the numbers of threads, which is explained by the limitations related to de decompression of the .7z file format. Except by PIGZ, algorithm ranking is maintained both during compression and decompression.

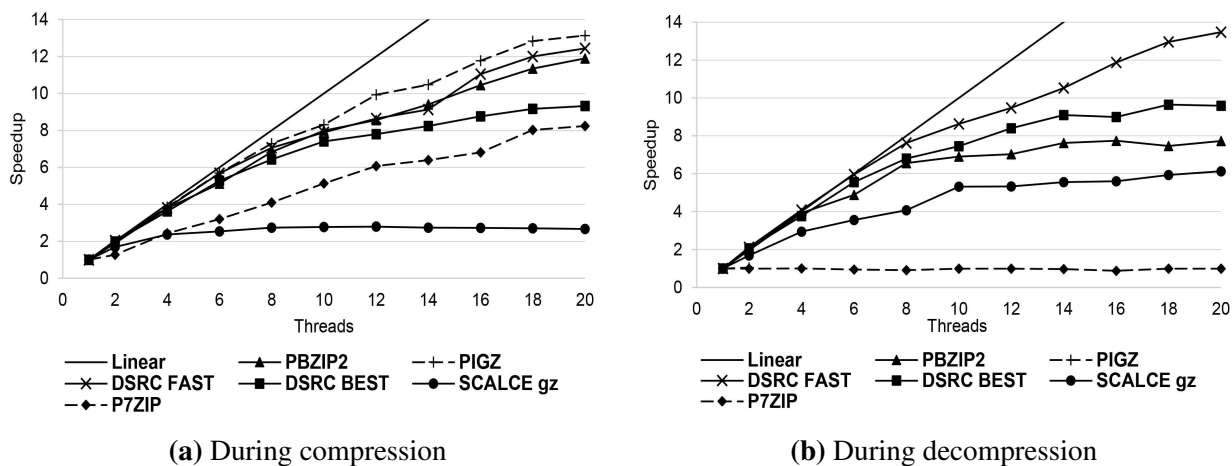
Best performance during multi-threaded experiments is resumed in Table 3-3. Notice how during compression, SCALCE gz multi-threaded performance is below DSRC-fast single-threaded, and in decompression is below PIGZ single-threaded runtime.

In Figure 3-4a we show the scalability of the programs in relation to the number of configured threads during compression. PBZIP2, PIGZ and DSRC-fast performed similarly, scaling close to linearly. PIGZ accelerated its performance up to  $13.1\times$ , DSRC-fast up to  $12.4\times$ , PBZIP2 up to  $11.9\times$  DSRC-best up to  $9.3\times$  and P7ZIP up to  $8.3\times$ , all using 20 threads. SCALCE gz improved only up to  $2.8\times$  using 12 threads. It is not clear why SCALCEgz had this low scalability (which remained almost constant from 4 threads on) since the authors do not provide any details about the parallelism model used to implement the boosting scheme, which is very important since we

**Table 3-3.** Summary of best multi-threaded performance results. Throughput (in MBps) reached in single-thread (Ts) mode, maximum throughput reached in multi-threaded (Tm) mode and the number of threads (N) required to reach this maximum. Results for compression (left) and decompression (right) are shown. Best results of each category are in bold.

Program	Compression			Decompression		
	Ts	Tm	N	Ts	Tm	N
<i>PBZIP2</i>	<b>8.13</b>	<b>96.68</b>	20	18.90	153.44	16
<i>PIGZ</i>	6.25	82.1	20	<b>130.6</b>	<b>178.96</b>	2
<i>P7ZIP</i>	1.0	8.24	20	49.50	49.50	1
<i>DSRC – best</i>	37.32	347.68	20	27.85	268.72	18
<i>DSRC – fast</i>	<b>54.14</b>	<b>673.26</b>	20	<b>61.07</b>	<b>822.64</b>	20
<i>SCALCE – gz</i>	9.8	27.36	12	14.3	87.71	20

presume it takes most of the runtime.

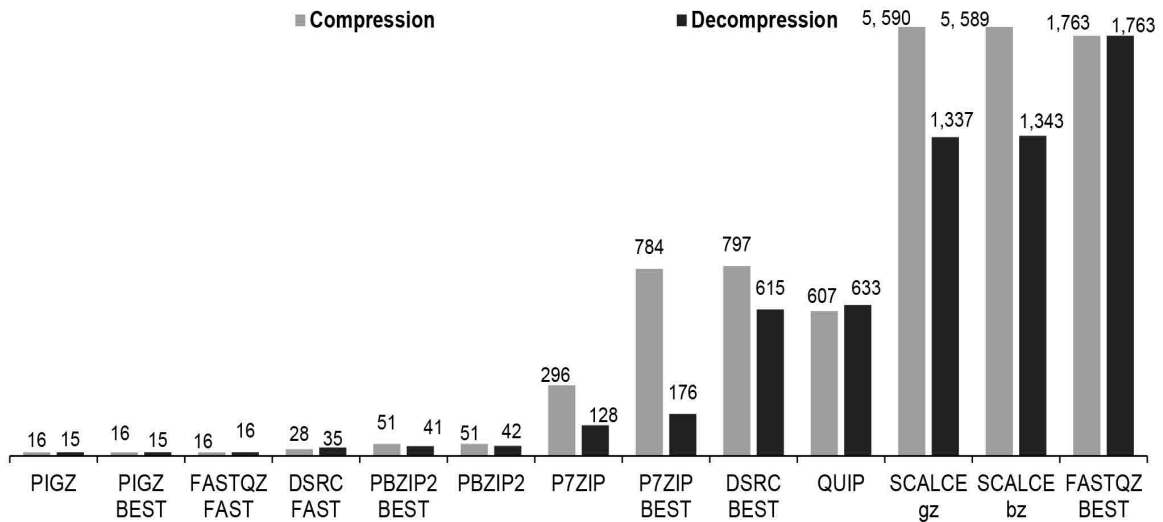


**Figure 3-4.** Speedup

Decompression speedup results are shown in Figure 3-4b. Only DSRC-fast kept increasing the speedup close to the linear case and performing 13.4× using 20 threads. DSRC-best improved up to 9.6× using 18 threads, PBZIP2 reached a maximum 7.7× improvement using 16 threads, SCALCE gz reached 6.1× (20 threads) and PIGZ improved its performance 1.06× using the extra thread. From 14 threads on, PBZIP2 performance remained almost constant. P7ZIP showed no improvement at all, for the reasons previously discussed.

### 3.1.3.4. Memory Usage

Maximum peak memory and maximum average memory used during execution were measured using Valgrind's `-pages-as-heap=yes` option, in order to get lower-level page profiling. Maximum peak memory represents the minimum memory that should be available for a program to run successfully. Results from single-thread tests are shown in Figure 3-5.

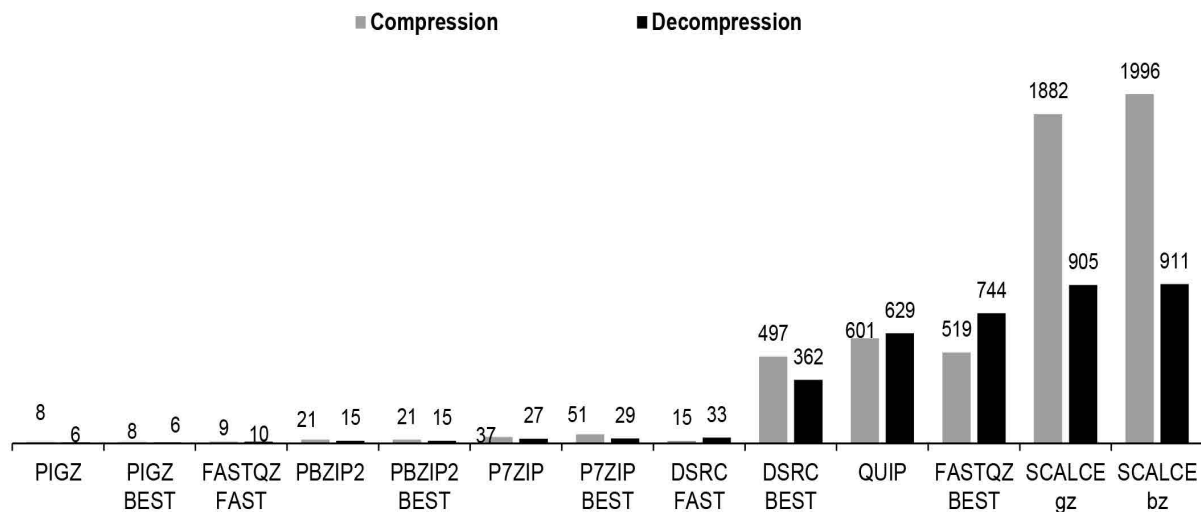


**Figure 3-5.** Maximum peak memory during single-thread execution (in MB), ordered by requirement during decompression

Requirements were similar during compression and decompression, except for DSRC-best, SCALCE and P7ZIP which demanded less memory during decompression. The most demanding tools in terms of peak memory were SCALCE and FASTQZ-best which required around 5.5 GB and 1.6 GB respectively, more than twice the amount required by any other tool. Peaks of PBZIP and PIGZ were tens of times lower than those presented by domain-specific tools, except by DSRC-fast which is as efficient as general-purpose programs in this extent. There were no difference in the requirements of the default and *best* configurations of SCALCE, PBZIP2 and PIGZ. During compression, SCALCE peak memory was very large in comparison to the input files size (more than half the size). It required almost  $4\times$  more memory than in decompression, as a consequence of the reads processing in the boosting method.

Figure 3-6 shows maximum average memory usage, reflecting memory consumption during the whole runtime. Requirements tended to be similar during both tasks for QUIP, DSRC-fast and for all the general purpose tools. Again, SCALCE (both configurations) was the most demanding application in this experiment; and DSRC-fast was the most efficient domain-specific tool, being comparable to general-purpose tools. Domain-specific tools (excepting DSRC-fast) demanded at

least  $15\times$  more average memory than any of the general-purpose tools. PIGZ and PBZIP2 were extremely efficient (being the less memory demanding), spending less than 25 MB in each run.



**Figure 3-6.** Maximum average memory in single-thread execution (in MB), ordered by requirement during decompression

P7ZIP, SCALCE and FASTQZ best had peaks considerable higher than the average requirements. The rest of tools showed a more uniform memory demand during the execution.

In both experiments (peak and average memory), the *best* configuration (highest compression) aims to increasing compression ratio at expense of using more memory. In DSRC this happens due to the creation of a text-processing buffer and a dictionary for quick search of matches. In FASTQZ this is due to the data structures required for the construction of a set of context models. In SCALCE, memory demand of both configurations is the same.

We also measured memory during multi-threaded executions and scaling up the number of threads (Figures 3-7a and 3-7b). During compression, peak memory requirements of DSRC-fast (2.8 GB) were always higher (nearly twice) than those of PIGZ and PBZIP2. DSRC-best (up to 22 Gb) peaks were (in most cases) over  $10\times$  higher than DSRC-fast, and are not fully shown on the plots in order to keep clarity in visualization. SCALCE gz peak memory is not shown for compression task (also for clarity), it varied slightly from 5.5 to 6 GB when scaling from 2 to 20 threads. P7ZIP (demanding up to 4 GB) required at least twice the memory demanded for any other general purpose tool. PBZIP2 and PIGZ were the most efficient tools in this test (both used a maximum of 1.6 GB).

During decompression (Figure 3-7b) PIGZ had the lowest values, running with less than 50 MB (2 threads). For the rest, the amount of memory used grew much more than expected: DSRC-fast

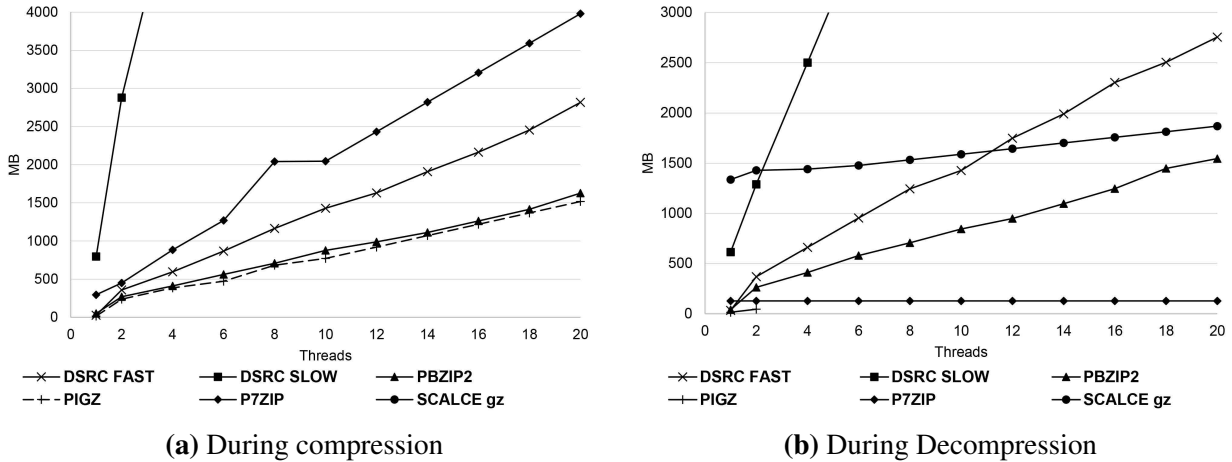


Figure 3-7. Maximum *peak* memory consumption (in MB) during multi-threaded execution

demanded around 2.7GB, DSRC-best up to 21.7 GB, PBZIP2 around 1.6 GB and SCALCE close to 1.8 GB. For P7ZIP the peak memory remained constant, which is explained because of the lacking of multi-threading during this task. SCALCE and PBZIP2 were the most efficient in terms of memory scaling during decompression.

Figure 3-8a shows the maximum average memory used during compression and Figure 3-8b during decompression, for the multi-threaded experiments.

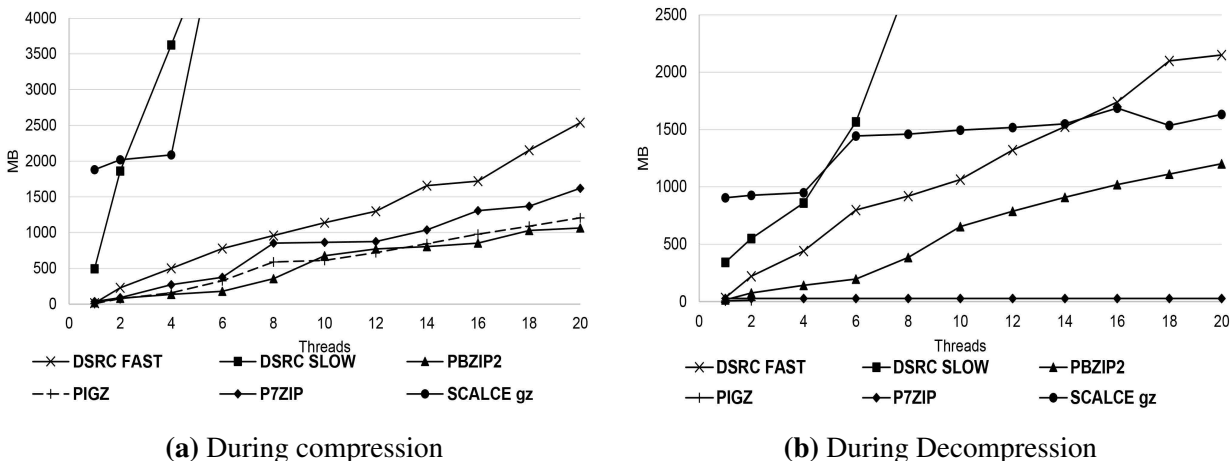


Figure 3-8. Maximum *average* memory consumption (in MB) during multi-threaded execution

For the average memory tests, DSRC-best was almost 9× worse than the fast version (in both tasks). DSRC-fast demanded around 2.5 GB during compression and 2.1 GB during decompression (using 20 threads in both tasks), DSRC-best over 18 GB (both tasks), PBZIP2 up to 1 GB during (both tasks), and PIGZ up to 1.2 GB during compression (20 threads) and only 10.5 MB during



decompression (for all threads). PBZIP2 and PIGZ performed similarly during compression, demanding almost half of DSRC-fast demands. Again, SCALCE demands during compression are not shown from 6 threads on (the varied between 5.4 and 5.7 GB, similar to the peak memory measurements), during decompression they were less than 1.6 GB. General-purpose tools handled memory in a more efficient way, except P7ZIP. P7ZIP demands grew faster as more threads were added during compression, asking up to 4 GB of memory. During the compression, P7ZIP memory requirements remained constant in 27 MB.

### 3.1.3.5. Discussion

Multiple metrics have been compared in order to analyze the presented methods. Even though there is no an absolute best compressor, the presented results may help the final user to select the most appropriate tool according to their priorities. It is clear that high compression ratios and fast performance are antagonizing goals. The slowest and more memory-demanding programs generally produced smaller compressed files. General-purpose tools presented compression ratios in the range of  $\sim 3:1$  to  $5:1$ , with compression rates up to 100 MBps (PBZIP2) and decompression rates of almost 180 MBps (PIGZ). In the case of domain-specific tools, compression ratios were between  $\sim 4:1$  and  $7:1$ , reaching up to 670 MBps during compression and 820 MBps during decompression (DSRC-fast). This allows to intuit that most of specialized tools used traditional compressors (same coding part), mostly optimizing the data-handling part and perhaps the modeling part.

DSRC-fast performed similarly fast in both compression and decompression, processing between 15 -25 MBps more than the closest competitor in each task. Even though its compression ratio was the lowest of all domain-specific tools (excepting FASTQZ fast), we should consider its FASTQ random access support as a significant plus for handling such large files. Additionally, we observed that the program showed the best speedup during multi-threaded execution (i.e. performance scalability), and the lowest memory requirements of all the domain-specific software evaluated. We did not see any significant gain in using the DSRC-best configuration, since it demanded a lot more computational resources with poor improvement, even running in several threads.

QUIP strategies achieved the third highest compression ratio but with little difference in comparison to the performance of FASTQZ and SCALCE in single-thread execution. Nevertheless its performance during decompression was  $3\times$  worse than DSRC-fast. It was also the third most memory-demanding tool.

SCALCE compression ratios (both modes) were slightly superior to QUIP, but this difference would only be worthy in a multi-threaded architecture and if there is priority in fast decompression times. We also noticed that despite being configured to use only 1 thread, SCALCE (both configurations) tended to execute a second thread occupying 20 -30% of an extra core during compression

and decompression tasks, so it was evident it did not run purely in a single-thread mode. SCALCE boosting method had the largest memory demands, occupying more than half the size of the input files.

Even though FASTQZ-best showed the highest compression ratio, its elevated runtime and memory requirements make it unusable when such metrics matter. According to our results, usage of FASTQZ-fast offered no benefit at all.

Regarding general-purpose tools, P7ZIP reached a significantly higher compression ratio than PBZIP2 and PIGZ, with a high performance during decompression but an extremely low performance when compressing. PIGZ tool had the lowest compression ratio and the lowest throughput during compression, but performed a lot better during decompression. PBZIP2 and PIGZ reflected a gradual improvement when scaling the number of threads, but PIGZ speedup was more efficient using additional hardware. Both tools were extremely efficient in memory management. P7ZIP showed an inefficient usage of memory and a poor harnessing of additional threads, scaling poorly during compression and having no improvement at all during decompression. This could be the main consequence of the complex combination of dynamic programming and Lempel–Ziv–Markov algorithms that P7ZIP implements.

Multi-threaded execution improved general-purpose tools performance up to  $13\times$  when compressing and almost  $8\times$  when decompressing. DRSC-fast had the best use of additional computing resources, improving up to  $13\times$  during both tasks using 20 threads; being up to  $7\times$  faster than the multi-threaded performance of the best general-purpose tool. However, in both cases this gain tended to decrease as the number of threads rose; most likely as a consequence of the increased communication overhead in comparison to the computation costs.

Multi-threading also demanded more memory than sequential runs. This phenomenon may be caused by the need for replicating data structures such as dictionaries when the (de)compression task is split among multiple threads. In general, parallelism not only improved overall runtime performance, but also increased memory requirements. In those cases, memory demands went from tens of MB for the sequential case to hundreds for the multi-threaded case, doubling or tripling the expected increase in memory-demand. In this context, DSRC-best, P7ZIP and SCALCE managed memory in a noticeable less efficient way than the rest of tools.

Our results are consistent with those independently reported about sequential performance in [22, 27, 29, 31, 32, 36, 105] and DSRC parallel performance in [51]. This experience provided a better understanding of the capabilities of non-referential strategies and stood clear the need of harnessing HPC resources for accelerating the compression of genomic data. Now, considering these results, the previous reports about referential genome compression and the theoretical potential of referential compression, the need of applying referential approaches for read sequences

compression in FASTQ format became clear.

## 3.2. State of the Art on Referential Compressors

After completing a detailed review on the current state of NGS data compression, it became evident that, by the time of such evaluation: (1) No referential compressor for FASTQ files had been presented, and (2) Performance of non-referential compressors was several times below from what a referential approach could theoretically achieve. Also, restrictions related to input data features (file size, read size, technology of the sequencing machine), excessive runtime or low compression ratios, have limited the usage and effectiveness of non-referential compressors. For those reasons we decided to base the work of this thesis on a referential compression approach. Although referential algorithms are not useful for compressing new species genomes, because of the need for a reference DNA sequence, they are still relevant given the envisioned studies that will sequence a huge amount of genomes of individuals in the near future [11]. Specific strategies must be proposed to solve the reference-selection issue.

Referential compressors for DNA sequences can be divided into different categories. Due to the scope of this research, we will group them according to the characteristics of the data to be compressed. The first two categories presented next will be discussed briefly, because our interest is focused on the third category: the referential compression of NGS data in FASTQ format.

As stated in 2.1.1, a whole genome represents processed data in a long sequence that could contain thousands of millions of bases. Read-sequences correspond to raw data represented in hundreds of million of short sequences (reads) in the FASTQ file. Both data are very different in size, redundancy and structure.

### 3.2.1. Genome Data Compression

Two approaches are considered here: the first one is focused on compressing a unique long sequence (a whole genome) [21, 23, 106–108, 108–111]. The second approach involves the compression of highly similar collections of whole genomes [112–117]. The level of redundancy in data implies applying different strategies in both cases.

In most of the reviewed works, the authors focused on how to store the differences between the read sequences and an external reference DNA string, such metadata is commonly called as alignment/mapping data. Commonly they consider three kinds of differences: deletes, inserts and mismatches. Also, the selection of the reference sequence has a dramatic effect in the performance of the algorithm, affecting the compression ratio in a range between 24 and hundreds of times for whole genome compression.

A very recent approach described in [116] uses a multi-level matching algorithm both in the compressor and the decompressor, achieving very high compression ratios. The authors state that they reached compression ratios four times greater than any other solution in a large collection of genomes (about 9200).

### 3.2.2. Compression of Read Sequences Along with Alignment/Mapping Information

It refers to compressing file formats that put together reads along with read-to-reference alignment data. This approach commonly takes the input from SAM/BAM files [118] and requires a specific compression approach to target the alignment information. Finally, such file formats as SAM/BAM have a significant amount of additional fields that should be compressed too. Some featured tools are: [40, 85, 93, 119–124]. Frequently, encoding techniques (Golomb code, Huffman, others) were used to represent the differences in alignment, performing lossless compression for reads that aligned to a well-studied reference sequence with few differences. In Goby [122], authors proposed combining multi-tier data organization (based on a referential approach) and a new file format as an alternative to SAM/BAM format. The tool has several objectives, including reducing the storage cost of large sequencing datasets. CRAM [40, 123] was also an alternative format proposed for the European Nucleotide Archive, for the compression of alignment information; it is currently part of the SAMTOOLS package [124]. Experiments have shown that the compression ratio could be twice better than the best traditional general compressor tested, which nowadays could be comparable to the performance of non-referential compressors [85].

Encoding alignment information is a field of our interest since we will use this approach for the referential compression. We found a very important antecedent in the work of Kozanitis et al in 2011 [42]; they introduced a set of domain specific referential lossless compression schemes for reads and alignment data. According to its authors, *Slimgene*, outperformed traditional compressors by  $6\times$  under restricted conditions. Nevertheless, results were promising and extensively cited. As far as we know, the development did not evolve.

### 3.2.3. Compression of Next Generation Sequencing Raw Data

In this category, we either find (a) multi-file compressors for large datasets of highly related reads or (b) single-file compressors for reads. In both cases, this implies handling short raw redundant sequences, and sometimes the compressor also process the IDs and Qs, although using different approaches. Examples of such compressors include [41, 45, 125].

In Kpath [70], authors combined path encoding, De Bruijn graphs and context-dependent arithmetic coding in order to offer reference-based compression without the need of a previous alignment. Authors claimed that a high compression could be achieved even if the reference was poorly matched to the reads. Reported results showed that the compression ratio was up to twice better than the best specialized non-referential compressors tested.

Authors of Leon [69] proposed the use of a probabilistic De Bruijn graph based on a Bloom filter, and then recording the reads and Qs as mapped paths in the graph using arithmetic encoding. Reported results showed that the compression ratio of the tool crucially depended on the quality of the reference, which is built from the reads. In overall, that compression ratio was up to 10% higher than the non-referential methods presented in that report.

Even though compressing the three data streams in FASTQ files (read sequences, IDs and Qs) is required for a truly lossless compression, few tools offer such capability. The well known non-referential compressors Quip [22], Fastqz [50] and Fqzcomp [50] are able to compress the whole FASTQ and are able to work in referential mode. However, the compression ratio achieved by those methods in the referential configuration, is not better than their own non-referential counterparts. Recent versions of Leon, compress all the data in a FASTQ file. In 2015, FQZip [126]) was presented as a reference-based method to compress the whole FASTQ file, and evolved to a second version of a light-weight mapping model (LWFQZip2 [75]), achieving compression ratios comparable to those of non-referential programs.

Although some FASTQ referential compressors [69, 75] have been presented reporting good results when compared to non-referential programs; we have discussed above the theoretical bases and related application of referential compressors that led us to expect higher compression ratios, leading us to believe that there is ample room for improvement in the development of referential compression algorithms for FASTQ files. Naturally, these expectations are limited by the compression of the other two streams in the FASTQ file: the IDs, and most of all, the Qs that span over a larger alphabet. On the other hand, it is clear that there will be more interest in the development and usage of referential compressors for FASTQ files if: (a) the step corresponding to the selection of an "appropriate reference" is conceived as part of the program tasks and (b) High Performance Computing (HPC) resources are used to reduce the running times. Considering these aforementioned challenges, our research falls within this category of referential compressors for FASTQ NGS data. Experimental and quantitative comparisons of the software in the state of the art in this field will be presented later in this document.

Due to the need for identifying the minimum amount of differences between each read and the reference, read alignment is a useful step in this type of compression. Also, due to the independence between successive reads, work on improving and parallelizing referential algorithms is pertinent, to reach a reduction in execution time.

### 3.3. Summary

Two trends have dominated the scene of FASTQ files compression: non-referential and referential approaches. In this chapter, we presented a comprehensive review and an experimental quantitative comparison of the non-referential compressors for FASTQ files, bringing a deeper knowledge about the state of the current publicly available software.

During our performance evaluation, it was clear that high compression ratios and fast performance are competing goals. The slowest and more memory demanding programs generally produced smaller compressed files. General-purpose tools, although not optimal, are easy to use alternatives in the absence of specialized software. Domain-specific tools take advantage of the intrinsic characteristics of the DNA raw data, achieving a more efficient compression, with compression ratios up to 70 % higher. Despite of the increase in runtime caused by the use of highly specialized methods, HPC parallel architectures have been leveraged to accelerate the compression and decompression tasks. Another high cost to pay for the benefits, is the large amount of memory consumed.

On the algorithmic side, it became clear that the most successful tools have developed hybrid approaches, combining different compression techniques. Powerful traditional techniques such as Huffman coding, dictionary approaches and others, are being combined with more modern techniques such as boosters, adaptive models, context models, Markov models, among other strategies. The dynamic combination of such techniques has led to the development of specialized software with a separated treatment for every stream inside a FASTQ file, bringing both higher compression ratios and faster execution. However, compression ratios achieved by top non-referential compressors were still below from what a referential compressor could theoretically achieve, exhibiting a great research opportunity. Despite the promising results shown by referential genome compressors (most of them based on alignment), no referential approach for FASTQ files compression had been presented until the end of 2014.

At the beginning of this chapter, our initial goal was to get a better understanding of the incipient and barely studied FASTQ compression field. Then, we studied and compared experimentally the most relevant compressors in the current state of the art, which involved only non-referential compressors since there were no referential compressors by then. Observing the compression limitations, we got hints for improvement. Considering the theoretical improvement commonly achieved by referential methods in general, and the compression observed in referential compressors for genomes, we decided to adopt the referential approach to tackle our main objective: the FASTQ reads compression. During our review of the referential compressor developed for related data, it became evident the strong dependency of a such approach in appropriate reference

to achieve good results, and the corresponding usability problem in the genomic field; since in bioinformatics an "appropriate reference" is a matter of discussion. In consequence, we decided to invest some efforts in the design of a framework to support the referential compression through automatic selection of the needed reference. Results from such effort are presented in the following chapter 4.

The work presented in this chapter has been partially published in:

**Guerra A.**, Isaza S.,and Lotero J. (2016). Performance comparison of sequential and parallel compression applications for DNA raw data. *Journal of Supercomputing*. Springer US, vol. 72, no. 12. ISSN online:1573-0484, printed: ISSN 0920-8542.

**Guerra A.**, Cabarcas F., Alzate J., and Isaza S.(2015). Herramientas de compresión de propósito general y específico aplicadas a secuencias genómicas. III Congreso Colombiano de Biología Computacional y Bioinformática (CCBCOL3). Medellín, September 2015.

## 4. A Workflow for Referential Compression of FASTQ

Our previous literature review showed that the main cause that has limited the wide adoption of the reference compressors lies in the need for an "appropriate reference" on which the whole compression process would depend. This reference (which usually correspond to a genome or pseudo-genome of several GB) has been commonly selected by an expert, considering a mostly biological criteria. However, it may not be available for organisms or species that have not been previously sequenced, or there may not be consensus in the biology community to define what can be considered an "appropriate reference", even for species with available assembled genomes. Such reference needs to have the minimum distance with the target data in order to improve the compression as much as possible. On the other hand, given the high storage cost of the references they will only be useful if a reference may be appropriate for the compression of multiple different input files.

In consequence, as a previous step to the development of our referential compressor we have designed a comprehensive workflow that covers the high level algorithm of the operational aspects of the compression process. In this chapter we present such workflow for FASTQ referential compression, which aims at providing the usability of non-referential compressors while achieving the potentially higher compression ratios of referential approaches. It includes the automatic selection of the reference, how the tasks are structured, how they are carried out, their order and synchronization, and the respective information flow. Although the whole workflow is partially implemented, we provide additional discussion about the main issues to be considered for the implementation of the inner blocks.

### 4.1. The Workflow Model

The workflow is designed to fill the main gaps found in the state of the art: automated selection of an appropriate reference and the specialized compression of each of the streams inside of the input FASTQ file.

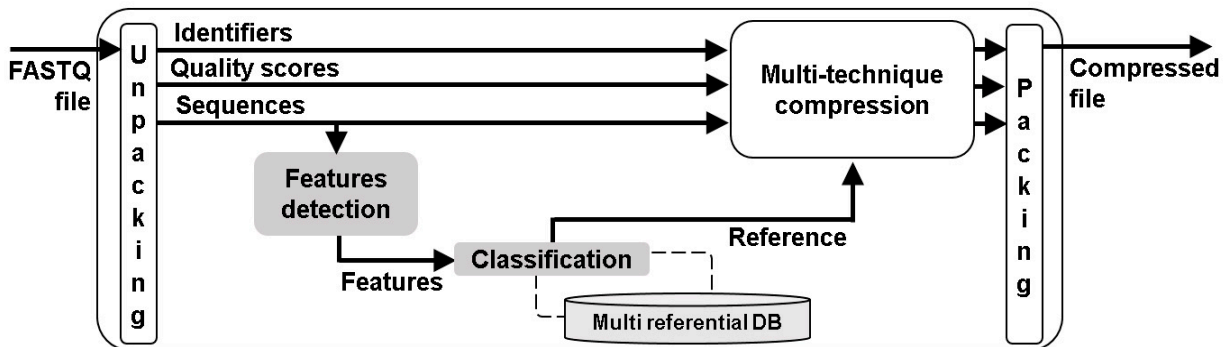
The design features multiple stages to fulfil such purposes, as shown in Figure 4-1. The content of the input FASTQ file is split so every data stream inside is compressed separately, in order to



provide a specialized compression that harness the different properties. At the end of the process the resulting compressed data are packed back together along with the metadata associated to the compression process (address of the respective reference, its length and the read sequences length).

A set of features is calculated from the input file, which are immediately used to select the most similar reference available in the database, which is passed to the referential compressor along with all the streams of the input file. The three main blocks are:

- (1) The characterization of the read sequences to detect relevant features,
- (2) the classification process that selects a corresponding reference sequence from the database, using the features resulting from previous step
- (3) the specialized compression of the three different data streams in the FASTQ file, focusing on the reads referential compression using the reference selected in (2).



**Figure 4-1.** Referential compression workflow: (1) Characterization of the read sequences through a features detection algorithm, (2) Classification of features to associate the input to a reference in the database (3) Multi-technique compression scheme for each stream of the FASTQ. Gray boxes are yet to be implemented.

Additionally, to decrease the impact of the references storage, we propose building a database of featured references that allows using them in many different compression processes.

Due to the high degree of specialization required for each of the tasks within the workflow blocks, in some cases we decided to use third-party software in order to perform the various workflow components. For the same reason, highly specialized blocks such as the feature detector and the classifier are only presented in a design stage, leaving their implementation as a future task. Instead, this thesis presents in chapters 5 and 6 the design and implementation of the core referential compression algorithm as well as the corresponding performance analysis and optimization through parallel computing.

## 4.2. Sequences Database

The design of the references database is an extensive research work in itself. This database must be created previous to the software usage, containing a number of representative genomes in FASTA format, corresponding to a wide set of organisms to increase the possible correspondence with any different input. FASTA format is a text-based format for representing nucleotide sequences, in which base pairs are represented using single-letter codes <sup>1</sup>. In practice, the user could add at least one genome entry for every different species he works with. Currently, the creation and storage of a references database is a matter of research that may include the construction of synthetic references through different methods: building pseudo genomes, using different chromosomes from different organisms (instead of a whole single genome), and even calculating common sub-strings for particular species. This is specially important to our research since such strategies could lead to create better references, with more statistically common data segments. Also, efficient representation of the genomes in the database is a topic of interest that has been dealt recently [127]. The database must also contain the data corresponding to the features calculated for each references, avoiding its calculation every time they are required by the classifier block.

There are many highly recognized public repositories for genomic data <sup>2 3 4 5 6</sup>. We selected the Sequence Read Archive (SRA) <sup>7</sup> as the main source of data, due to its impact and recognition in the bioinformatics community.

Since in this document the blocks for automated selection of the references are only presented conceptually, there was no need of an extended database including many references. For that particular reason, the current references database is highly related to the FASTQ files in our test datasets, according to biological criteria (as specified in the SRA). When all stages of the workflow are fully implemented, this strict biological correspondence between entries and references will not be necessary.

The dataset for tests was built considering plants, bacteria and human genomic FASTQ files obtained with Illumina platforms; along with their corresponding references. A detailed description of the files in the final dataset will be presented before each test. We selected files with a size between 1 Gb and 60 Gb, and fixed size reads length from 36 to 151 bases. Every file was downloaded along with the corresponding reference, preferring files already used in relevant reports in the state of the art. A comprehensive documentation of each file was done considering important

<sup>1</sup><https://zhanglab.ccmb.med.umich.edu/FASTA/>

<sup>2</sup><https://www.ncbi.nlm.nih.gov/genbank/>

<sup>3</sup><http://www.internationalgenome.org/>

<sup>4</sup><https://www.ebi.ac.uk/ega/datasets>

<sup>5</sup><https://www.ebi.ac.uk/ena/>

<sup>6</sup><https://www.ddbj.nig.ac.jp/index-e.html>

<sup>7</sup><https://www.ncbi.nlm.nih.gov/>

meta-data as: file identifier, experiment identifier, organism, file size, read length, amount of reads, coverage, among other specialized fields. We developed algorithms to process the files to adapt them to our entry conditions.

### 4.3. Packing and Unpacking

The three data streams in a FASTQ file are very different in content, length, alphabet and the level of similarity among different reads; in consequence, differentiated compression strategies must be applied in each case. In order to set things up for the separate processing of data streams, two blocks are placed at the beginning and at the end of the workflow. The *unpacking* block reads the FASTQ file and creates three data streams for further processing. The *packing* block joins and stores in a single file the three compressed data streams with all the metadata needed for an effective decompression.

### 4.4. Sequences Features Detection

To select the most appropriate reference for an input it is necessary to find the minimum distance between such reads and each of the candidate references in the database. Since both the reads in the FASTQ input and the references in the database contain large amounts of data, calculating this similarity is a task that requires lots of computing resources. Therefore, we propose to describe them through a set of specific features that reduce their size and allow performing an accurate classification within acceptable execution times.

References have been usually selected based on a biological criteria, but we propose to consider a mathematical approach that improves the subsequent referential compression. The main goal is to find the reference with the highest degree of local similarity between the maximum amount of reads and the reference, according to some mathematical metric.

Previous works have demonstrated how methods that have been traditionally applied for text analysis: k-mers, suffix arrays or FM-index [74, 128–131] are useful for DNA features detection. Those strategies are oriented to reduce computing times and memory usage. A recent approach proposes self-calculating the reference applying De Bruijn graphs over the input sequences [69].

Features must be selected carefully, since the effectiveness of the classification to be performed in the next step, and the further compression, will depend on them. The main objectives that must be considered when selecting the features are:

- To avoid over-adjustment and redundancy between attributes.
- To reflect the most information underlying in original data.
- To provide effective attributes with the lowest possible computational cost.
- To represent measures directly related to the local distance between reads and reference.

Many methods have been proposed for the features selection in general. Most of them are based on the comparison of classical statistics as means, variances and standard deviations or other advanced univariate or multivariate statistical indicators that allow to quantify how one or more attributes allow to discriminate between two or more classes. This has been the basis of class separation methods (linear or preferably non-linear) as well as various types of clustering algorithms, the relief-f (symmetrical Tau) algorithm, and other approaches for the bisection of the feature space [132–138].

Two main categories of the proposed methods can be recognized in the literature reviewed—methods based in word frequency, and those that do not require analysing the sequence through fixed word-length segments. The first are the pattern-based features, which includes procedures based on metrics defined in coordinate space of word-count vectors, such as the Euclidean distance, Hamming distance, or relative entropy of frequency distributions. On the contrary, the second category corresponds to techniques that are independent from the resolution of the sequence, i.e. they do not involve counting segments of fixed length. They include the use of Kolmogorov complexity theory and scale-independent representation of sequences by iterative maps. These two categories of methods have distinct theoretical lineages and an unequal amount and variety of techniques explored in the published reports, far fewer for the latter [139].

Considering the wide range of options, final selected features must represent diverse information about data. We suggest to select them considering:

- a. Features that allows transforming the read sequences into a feature vector.
- b. Features that contain distance based information; this may include specific words of interest as a reduced set of the  $k$  most common longest sub-strings.

The challenge of applying pattern-based features selection on symbolic sequences is how to efficiently search for the features satisfying the criteria. When selecting the final features, special attention must be paid to a recently proposed structure called FM-index, which allows calculating efficiently many string based features (such as longest common substring). This structure is very relevant in this research, due to the fact that its content is used also for other tasks that are part of the proposed workflow, hence it could be reused as much as needed, without consuming additional

computational resources. A detailed explanation on the concepts and usage of the FM-Index in this research will be presented in chapters 5 and 6.

Detailed work on describing and analyzing the state of the art on features detection for DNA analysis has been reported in [140–146].

## 4.5. Read Sequences Classification

This block takes the features of the read sequences calculated in the previous step and determines the most appropriate reference, the one with the highest level of local similarity with the input reads. Local similarity is a metric calculating the highest amount of reads having the least differences with the reference. This is better than an approach of global similarity in which short sub-strings inside different reads are more similar to the whole DNA string. As we stated before, we are interested more in mathematical distances than in applying a biological criteria.

It has been widely shown how the use of learning machines and intelligent algorithms has been applied successfully in tasks of multi-class pairings. Diverse classes of artificial neural networks and other machines such as support vector machines are currently used for this task. In the case of DNA classification there have been some relevant works [142, 145, 147–151]. Some general purpose implementations have showed good performance in a wide range of problems <sup>8</sup>.

We propose a combination of two successful sequence classification methods [142]:

(a) Feature based classification, which transforms an input sequence (reads or reference) into a feature vector (calculated in the previous step of the workflow) and then applies conventional classification methods; such as decision trees, support vector machines and neural networks, designed for classifying feature vectors.

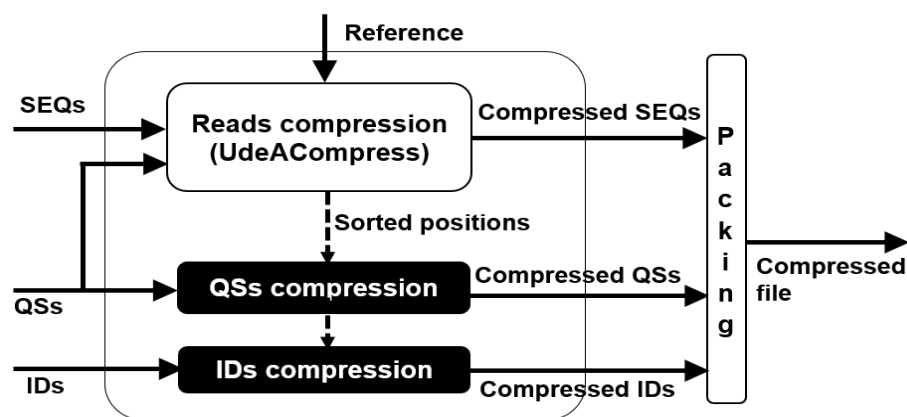
(b) Sequence distance based classification. A distance function must be designed, which measures the similarity between sequences. The idea is to use this second criteria to ponder the quality of the classification performed in (a).

## 4.6. Multi-Technique Compression Scheme

Actual compression happens in the multi-technique compression block. The multi-technique feature means the compressor uses specific algorithms for the different data streams as it will be

<sup>8</sup><https://www.csie.ntu.edu.tw/~cjlin/libsvm/>

discussed in this section. The main internal components of this block are shown in Figure 4-2. This approach allows to maximize the use of the specific properties of each of the streams to be compressed.



**Figure 4-2.** The multi-technique compression scheme: Referential compressor for reads sequences, identifiers (IDs) compressor, quality scores (QSSs) compressor. Currently, black boxes are implemented using third-party software.

QSSs play an important role in our strategy for the compression of read sequences and, in consequence, they are also fed to the reads compression block. Also, at the bottom of Figure 4-2 it can be observed that the compression of IDs and QSSs is performed once the sorted positions of all the reads have been computed. This guarantees an adequate decompression of the reads in the input data, since the compressor sorts the read sequences before compressing them. Both implementation issues will be explained in the chapter 5.

#### 4.6.1. Lossless Compression of Read Sequences

At the beginning of this research, the referential compression had been successfully applied for the compression of whole genomes; the most relevant strategies of referential compression used mapping or read-to-reference alignment information [23, 106]. Sequences alignment basically consist of arranging two strings of characters in a way that the maximum position-to-position similarity is achieved.

However, in our review of the state of the art we did not find any report of referential compression of NGS raw data. The closest work was the strategy applied by Kozanitis [42] for the compression of short sequences, through the available alignment information according to the Illumina export file format. For that reason we decided to apply an alignment based approach in the FASTQ reads compressor. Many challenges had to be solved to achieve such goal, due to the differences in size,

redundancy and structure of NGS data in FASTQ format.

FASTQ read sequences are compressed with UdeACompress, the referential lossless compression algorithm we developed and the core of the proposed workflow. To describe the development, implementation and performance evaluation of UdeACompress is the main objective of chapter 5.

### 4.6.2. Identifiers Compression

IDs have a format that depends completely on the sequencing platform. They use a wider alphabet and are a few times shorter than the other streams in the FASTQ. This is an example of a FASTQ identifier according to recent Illumina platforms:

```
@SIM:1:FCX:1:15:6329:1045 1:N:0:2
```

which obeys to the following structure:

```
@<instrument>:<run number>:<flowcell ID>:<lane>:<tile>:<x-pos>:<y-pos> <read>:<is filtered>:<control number>:<sample number>
```

In such structure, it can be observed that some fields show little variation between contiguous reads within the same file, which is usually exploited using delta encoding approaches [152]. Table 4-1 presents the details about the meaning and the usage of each field<sup>9</sup>.

In the research priorities we had established that the first objective to be attacked was the referential compression of the reads sequences, since this had not been done previously. In turn, in the context of non-referential compressors, good efforts had been made for the compression of IDs. For that reason, the compression of IDs and Qs was initially meant to be performed using third-party software. The final decision about such software selection is presented next, along with Qs compression.

### 4.6.3. Quality Scores Compression

Qs are more difficult to compress than IDs. They have the same length of the read sequences but use a much larger alphabet of ASCII values that also depends on the sequencing platform; every read within the FASTQ file has its own corresponding string of QS values. There is currently research concerned specifically on Qs compression [153, 154].

<sup>9</sup>[http://support.illumina.com/content/dam/illumina-support/help/BaseSpaceHelp\\_v2/Content/Vault/Informatics/Sequencing\\_Analysis/BS/swSEQ\\_mBS\\_FASTQFiles.htm](http://support.illumina.com/content/dam/illumina-support/help/BaseSpaceHelp_v2/Content/Vault/Informatics/Sequencing_Analysis/BS/swSEQ_mBS_FASTQFiles.htm)

**Table 4-1.** Illumina's identifiers structure

Field	Requirements	Description
@	@	Each sequence ID line starts with @
<instrument>	String: a-z,A-Z,0-9,-	Instrument ID
<run number>	Numerical	Run number on instrument
<flowcell ID>	String: a-z,A-Z,0-9	Flowcell ID
<lane>	Numerical	Lane number
<tile>	Numerical	Tile number
<x_pos>	Numerical	X coordinate of cluster
<y_pos>	Numerical	Y coordinate of cluster
<read>	Numerical	Read number: single read (1) or paired-end (2).
<is filtered>	Y or N	Y if the read is filtered (did not pass), N otherwise
<control number>	Numerical	0 if no control bits are on, an even number otherwise
<sample number>	Numerical	Sample number from sample sheet

The QS of each base in the read sequence, also known as a Phred or Q score, is an integer value representing the estimated probability of an error in an specific base of a read during sequencing, i.e. that such base is incorrect [46]. If P is the error probability, then it is calculated using the equation 4-1 [155].

$$P = 10^{-\frac{Q}{10}} \quad (4-1)$$

Qs are often represented as ASCII characters. The rule for converting an ASCII character to an integer may vary, it generally is  $Q = \text{ASCII\_CODE} - \text{ASCII\_BASE}$ . Here, ASCII\_CODE is the ASCII code for the character as found for each base in the FASTQ file; and ASCII\_BASE is a constant (commonly 33 or 64, according to the corresponding specific encoding). ASCII\_BASE 33 is now almost universally used.

The main object inside of the FASTQ file are the reads sequences. Even important, Qs are just metadata of the same size of reads, but defined over a very different alphabet and with very different specific properties (i.e redundancy, coding, among others). As mentioned before, an independent research line has appeared in order to compress them with specialized approaches. In consequence, we have focused on developing a referential compressor for the read sequences stream. To select which of the existing software could be useful in the compression of IDs and Qs we evaluated the performance of two of the most efficient programs in the state of the art for IDs and Qs only.

Programs were chosen considering their approaches for compressing specifically this type of streams, their efficiency, speed and software dependencies, according to our previous tests. From all the previously tested software DSRC [51] and Quip [22] matched our requirements for both compression tasks (IDs and Qs). Details about specific techniques used by each selected pro-



gram were presented previously, in chapter 3.1.1 of this document. Tests were performed over the same dataset that will be described in chapter 5.2.1 and including additionally 3 human FASTQ files. Results are shown in Table 4-2.

**Table 4-2.** Quip and DSRC compression ratio on FASTQ identifiers and quality scores

Species	QUIP ID	DSRC ID	QUIP QS	DSRC QS
<i>Bacteria</i>	93.7%	92.0%	83.8%	80.8%
<i>Human</i>	98.9%	97.0%	78.3%	72.6%
<i>Plants</i>	93.8%	92.4%	83.1%	79.5%

Compression ratios for IDs and QSs were similar for both tools, with no significant difference. We also measured running times for compression and decompression of each stream: performance was very similar for IDs; while for QSs DSRC was at least 30% faster during both tasks. Quip did not have any library dependency, while DSRC execution depended on the Boost libraries<sup>10</sup>.

Considering the results, we decided to use QUIP software version 1.1.8, which is implemented in ANSI C99. QUIP compresses the IDs using delta encoding and QSs using an order-3 Markovian models QSs in correlated positions [22].

## 4.7. Summary

Referential compressors have been successfully used for the compression of genomic sequences. However, some challenges prevented the usage of such approach for the compression of reads in FASTQ format. One of the main reasons for the limited usage of such strategies is the need for an appropriate reference that improves the results of the compression.

In this chapter, the design of a three-stages workflow oriented for the automated selection of an appropriate reference for each input has been presented. The first stage aims to calculate a set of features describing the reads in the FASTQ input file, which are subsequently used in the classification stage to establish correspondence with any of the pre-existing references in the system's database. The third and finally stage performs a multi-technique compression over each of the streams in the input. Since by the time we began this research there were no referential compressors for reads-sequences in FASTQ files, this was the obvious choice to focus the development.

Additionally, we have discussed the general guidelines that must be considered when creating the reference database and when designing the final structure of the stages I and II of such a workflow.

<sup>10</sup><https://www.boost.org/>

---

Finally, the basic issues related to the compression of IDs and Qs within the FASTQ file have been presented, tasks which are currently carried out by third-party software.

In the next chapter we will present the details of the design and implementation of the third block of the proposed workflow, the referential compressor for the read sequences; which has been the main focus of this thesis.

The work presented in this chapter has been partially published in:

**Guerra, A.**, Lotero, J., Aedo, J., and Isaza, S. (2019). Tackling the challenges of FASTQ referential compression. *Bioinformatics and Biology Insights*, volume 13, SAGE.

Lotero, J., Benavides, A., **Guerra, A.**, and Isaza, S. (2018). UdeAlignC: Fast Alignment for the Compression of DNA Reads. In *2018 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE.

## 5. UdeACompress: A Referential Compressor for FASTQ

The results obtained from the comprehensive analysis of the state of the art led us to identify not only the need for a workflow to improve the usability of referential compression, but also the relevance of developing new approaches to improve the compression ratios of existing specialized compressors.

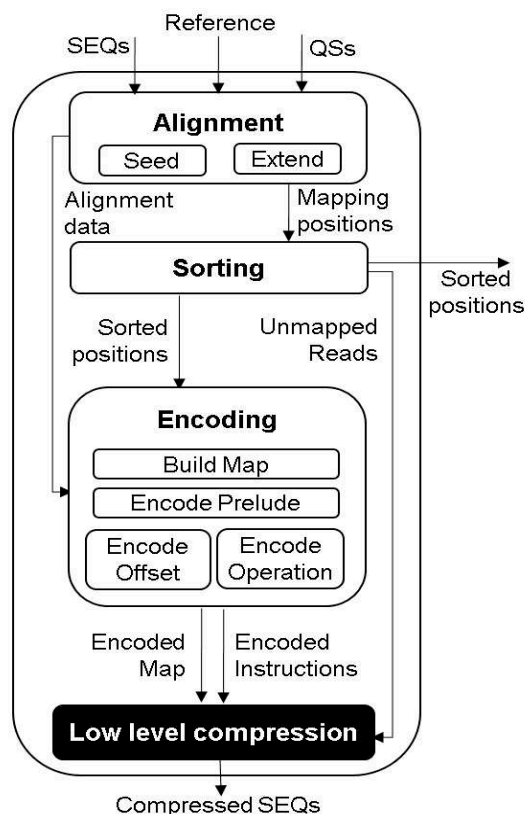
This chapter describes the implementation of the third and main stage of the workflow presented in chapter 4, a referential compressor for read sequences. The compressor is called UdeACompress, which is designed to improve the compression ratio through a referential approach based on finely encoding and compressing the results of a specialized alignment.

Finally we present a set of tests performed using real and simulated data, in order to evaluate and compare the performance of our compressor under different conditions.

### 5.1. Read-Sequences Compression with UdeACompress

UdeACompress performs a referential compression of the read sequences as the core of the multi-technique compression scheme. Such approach is based on the hypothesis that encoding the differences in the alignment between each read and the reference is a powerful strategy for referential compression. The approach aims to increase the quality of the performed alignment according to specific compression goals, and to improve the encoding and compression efficiency which results in higher compression capabilities.

UdeACompress first performs a specialized alignment between the input reads and the reference, and then sorts the reads according to their mapping position. These positions are encoded into a binary map and the alignment data is binary encoded. Finally, as some reads do not align to the reference, they are compressed separately using a low level compressor. The inner structure of this module is presented in Figure 5-1 and explained in detail in the coming sections.



**Figure 5-1.** UdeACompress block diagram. (1) Specialized read-to-reference alignment, (2) reads sorting, (3) reads encoding and (4) low level compression for encoded data and unmapped reads. The sorted positions resulting from the *sorting* step are used for the compression of identifiers and quality score as well, in order to guarantee a correct decompression. Currently, black boxes are implemented using third-party software.

### 5.1.1. Read-to-Reference Alignment

Sequence alignment is the process of arranging two strings of characters in a way that the maximum position-to-position equivalence is achieved. This procedure may involve splitting and separating segments of both strings, also denoted as creating *gaps*, to maximize the number of similarities between the two sequences. The optimum alignment describes the arrangement in which the least number of changes in the strings are applied to achieve the maximum similarity. The process of read alignment aims to calculate the least number of differences between a read and the reference, for its further encoding and compression. Sequence alignment is a core task within some referential genome compressors.

Read alignment is commonly applied in sequencing pipelines to efficiently map read sequences to a comparatively large genome, identifying the optimum position in which both sequences match, keeping track of differences between the reference genome and the sequenced sample in form of

mismatches, insertions and deletions [156], the latter two operations are also known as gaps. This process means millions of iterations in the majority of cases.

In UdeACompress, we have conceived an aligner that aims at finding an approximate matching to maximize a similarity score according to the compression goals. It is based on one of the most effective strategies when aligning short sequences: the seed-and-extend strategy [157]; which provides noticeable performance and accurate results. Such alignment is not based on a biological criteria but on reducing the distance between the aligned sequences, according to an optimality criterion that will be discussed in subsection 5.1.3.2. Multiple substrings are extracted from the read as potential seeds for the alignment. After an exact match is found through an FM-index [158, 159] that privileges bases with high QSs, an alignment is extended in both left and right directions using a modified Needleman-Wunsch algorithm [160]. The implementation of UdeAlignC used of the Succinct Data Structure Library (SDSL) [161]. The SDSL library is a comprehensive and widely referenced library which implements a variety of succinct data-structures, including suffix arrays and FM-index, and as long as a set of methods for storing, traversing and seeking information inside such structures. Additional details on the implementation and optimization of such aligner are presented in the research work Jaime Lotero [162, 163] and will not be addressed in this document.

The aligner receives the read sequences and QSs in the FASTQ file, along with the reference in FASTA format; and outputs a set of *instructions* describing the transformations needed to obtain the original read from a specific position in the reference. Those transformations (named mutations) are represented with several alphanumeric fields in SAM notation style, commonly used to express alignments. In such abstract notation, data is distributed among several fields which spans over a wide alphabet, containing: positions for every mapping, the direction/sense of the matchings and details about each of the mutations to be performed: offsets (displacement between changes), the type of operation that must be executed, and the corresponding target base. In certain cases those fields must be analyzed together to get unambiguous and precise information.

The few reads that cannot be aligned to the reference are passed as they are. We call them unmapped reads.

### 5.1.2. Reads Sorting

Sorting reads is a feature that some applications have applied to increase effectiveness in sequence compression [29, 53, 74], rearranging reads to take advantage of specific data features (commonly certain level of similarity). Although the original order is lost, it does not affect the later usage of the uncompressed FASTQ since their originally placement is anyway arbitrary [53].

After the alignment, UdeACompress must sort the reads (along with the alignment instructions) according to their mapping positions, as a pre-requisite for the upcoming encoding. To avoid moving such big data structures, we assign consecutive indexes to identify each read, and UdeACompress only sorts such indexes. Sorted indexes are used not only for encoding the alignment instructions, but also for compressing the IDs and Qs in that exact same order.

Sorting algorithms have been a matter of research per decades and it was not within our focus. We implemented and compared an optimized recursive quicksort and a LSD radixsort [164], both algorithms are very well know in literature for its efficiency when sorting unsigned integers [165–167]. The latter performed significantly faster (at least 30%). Basic structure of radixsort is presented in Algorithm 1.

---

**Algorithm 1** Reads Sorter
 

---

```

1: procedure LSD_SEQUENTIAL_RADIXSORT((unsigned Ind, size n, unsigned MapPos )
   ▷ n: number of reads
   ▷ Ind: read indexes
   ▷ MapPos: Mapping position of each read
2:   MaxLength ← MaximunDigitLength(Ind)
3:   for each Digitj until MaxLength do ▷ According to the number of digits to be evaluated
   per iteration
4:     Empty_Buckets(Buckets)
5:     for each Digitj until MaxLength do ▷ Build Histogram
6:       Buckets[CurrentDigit(Ind[i], i)] ++
7:     end for
8:     ExclusivePrefixSum(Buckets) ▷ Exclusive Prefix Sum
9:     for each Indi in Ind do ▷ Placing elements in their relative position
10:      aux ← CurrentDigit(Ind[i], j) ++
11:      Buffer[Buckets[aux]] ← MapPos[i]
12:      Buffer[outIndexes[aux]] ← Ind[i]
13:     end for
14:     SwapPointers(MapPos, Buffer)
15:     SwapPointers(Ind, outIndexes)
16:   end for
17: end procedure

```

---

The method iterates according to a fixed number of digits to be processed at the time. Then, it performs three main steps iteratively: building a histogram, exclusive prefixing and placing indexes into an auxiliary sorting memory location (Buffer) which is swapped at the end to update the input of the next iteration. The histogram and the placing steps depend on the amount of reads

in the input ( $n$ ), and the process is repeated according to the maximum number of digits of the biggest index ( $k$ ); which leads us to a complexity of  $O(n * k)$ , tending more precisely to  $O(n)$ . This method is simple and intrinsically parallel, which is useful for future optimizing goals (see section 6.2.3).

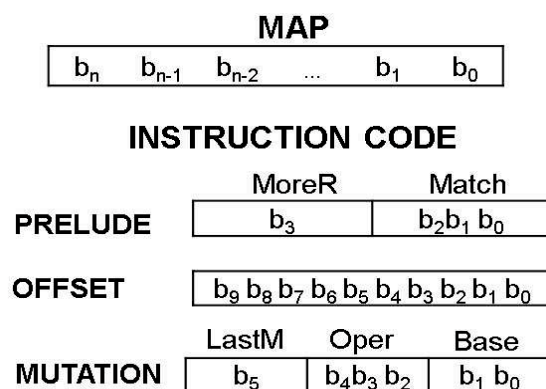
After this process, mapped and unmapped reads are separated, the alignment instructions are binary encoded and the unmapped reads are compressed in the low level compressor.

### 5.1.3. Encoding

The encoder in UdeACompress produces a space efficient binary coding of the alignments and was inspired by the work of Kozanitis [42] in encoding SAM data.

Two data structures form the code: a binary *map* of the alignment positions and the *instructions* array. This approach is conceived for files with fixed length reads, which is becoming a trend in sequencing machines [29, 168]. The code is also designed to be further compressed in the so called low level compressor block.

The code format is shown in Figure 5-2 and the algorithm designed to produce such encoding is shown in 2. The next subsections explain the details.



**Figure 5-2.** Instruction encoding. (I) A single map with as many bits ( $n$ ) as bases in the reference to indicate the matching positions. (II) A three fields instruction for each read: (a) a mandatory 4 bits PRELUDE for describing the matching types, (b) a 10 bits OFFSET to position each mutation and (c) the description of the MUTATION itself (6 bits). The OFFSET and MUTATION fields are not required for exact matching types.

**Algorithm 2** Instruction Encoder

---

```

1: procedure INSTRUCTION2BINARY(AlignmentInstructions[], n)
  ▷ n : number of reads
  ▷ output binary arrays: Map: alignment map, Preludes: preludes
  ▷ BinInst: encoded offsets and mutations
2:   Indexp ← 0
3:   Indexj ← 0
4:   Indexm ← 0
5:   for each Readi do
6:     UpdateMap(Map, MappingPositionReadi, Indexm)
7:     MoreFrags ← ((MappingPositionReadi = MappingPositionReadi + 1) and (i < n))
8:     Preludes[Indexp] ← PRELUDE(MoreFrags, AlignmentInstructions[i])
9:     Indexp ++
10:    for each Mutationk in Readi do
11:      BinInst[Indexj] ← OFFSET(AlignmentInstructions[i], k)
12:      Indexj ++
13:      BinInst[Indexj] ← MUTATION(AlignmentInstructions[i], k)
14:      Indexj ++
15:    end for
16:  end for
17: end procedure

```

---

**5.1.3.1. Map Building**

The *map*, shown in Figure 5-2 with an example reference above, is a binary array with as many bits as the reference. It only has 1's in the positions where one or more reads map (the start of an alignment). This map definition aims to reduce the cost of representing the mapping positions of the reads in the reference, and is the reason why the reads must be previously sorted. No matter how many reads there are in the input, the map size only depends on the reference length and is shared by all reads.

**5.1.3.2. Encoding of alignment instructions**

It is focused on generating a succinct representation of the alignment instructions in a binary space, while also producing a uniform distribution of bits in order to benefit more from the low level compression.

The first of the three fields that forms the code of every read is a mandatory and fixed sized PRELUDE as shown in Figure 5-2. It uses four bits for storing the matching information per read and



**Table 5-1.** Matching codes

Code	Type	Direction
000	Exact	<b>Forward:</b> A matching from left to right.
100	Approximate	
001	Exact	<b>Reverse:</b> The matching string is inverted.
101	Approximate	
010	Exact	<b>Complement:</b> Forward, with each base complemented.
110	Approximate	
011	Exact	<b>Reverse Complement:</b> Each base is complemented in a reverse match.
111	Approximate	

it is the minimal representation for a read matching in this model. The first bit of the PRELUDE indicates whether the next sorted read maps to this position as well, or not (MoreR). The next 3 bits encode 8 different kinds of matchings according to Table 5-1 (Match). The basic matchings (Forward and Reverse) could be exact or approximate (with at least one mutation), for a total of 4 cases. Since exact matchings are the cheapest to store, the strategy to increase its probability of occurrence was using an extra bit to incorporate two additional types of matchings well known in bioinformatics, but not commonly used in alignment: complement and reverse complement. This will also help to increase the amount of mapped reads, which are compressed more efficiently than the unmapped ones.

In complement matchings, each of the bases in the original sequences must be substituted by its biological complementary base (see Table 5-2), with N's not having a complement.

**Table 5-2.** Bases Complement

Base	Complementary Base
<i>A</i>	<i>T</i>
<i>T</i>	<i>A</i>
<i>C</i>	<i>G</i>
<i>G</i>	<i>C</i>

The prelude is enough for storing the exact matches, but efficiently storing the mismatches in the approximate matchings is the tricky part. The other two fields in the instruction coding are used for that purpose: OFFSET and MUTATION. These two fields only appear in the coding of reads that present alignment mutations.

The 10 bits OFFSET represents the shift between the last mutation (or the beginning of the read sequence if there is no previous mutation) and the place where the current mutation starts. The 10 bits reserved for the offset guarantee the capability of UdeACompress to support reads of up

to 1024 bases. In consequence, this field has the highest storage cost and will perform best when compressing reads of size 1024.

There is a current trend in sequencing technologies to produce larger reads [169]. Larger reads increase overlap between reads, and more overlap with your reference sequence. In consequence it is easier to put (assemble) the sequence back together. The most relevant companies already have released sequencers able to produce reads with several hundreds of bases (Illumina, Ion, Roche 454)<sup>1</sup>.

The third field of the instruction is called MUTATION as it describes in detail the type of transformation required to obtain the read from the reference. We use the first bit of this field to indicate whether this is the last MUTATION of the read (LastM). The next three bits describe the operation (Oper) to be applied. We defined 8 different types of operations (see Table 5-3) based on the mismatches and gaps commonly used in bioinformatics: substitutions, deletions and insertions. Finally, we use the last two bits to express the base required to perform the operation (Base). Since we only had two bits to express five possible base values (A, C, G, T, N), we store the distance (to be precise, the *distance - 1*) between the base in the reference and the target base in the read, according to the scheme in Figure 5-3.

	A	C	G	T	N
A	-	1	2	3	4
C	4	-	1	2	3
G	3	4	-	1	2
T	2	3	4	-	1
N	1	2	3	4	-

**Figure 5-3.** Circular base distances scheme.

Bases are needed only for single insertions and for substitutions, but in some cases of insertions the target base may not correspond to a base in the reference (e.g in insertions at the beginning or at the end of the reference); hence representing base distances is not possible. To overcome this issue, we separated the regular bases insertions (with target bases: A, C, G, T) from the case of N insertions (which are more common) as different operations. In this scheme, for regular bases insertions (from now on, insertions), the number in the base field represents directly the letter of the target base to be inserted; and for insertions of N (from now on, Ninsertions) both bits can be omitted or be set to zero.

Clearly, the more mutations per read, the lower the compression ratio. Therefore, we applied the following strategies:

<sup>1</sup><https://genohub.com/ngs-instrument-guide/>

a) Selecting the least possible number of mutations in the alignment: We influenced the aligner so that an exact matching is always selected if possible. If there are several different matchings, the one with least mutations is selected. Additional matchings previously introduced aim to achieve this goal as well.

**Table 5-3.** Probability of mutations occurrence

Type of Operation	Probability
Single Substitution	0.63
Single Deletion	0.15
Insertion (any base but N)	0.071
Contiguous Deletion	0.065
Ninsertion (N's only)	0.049
Triple Contiguous Deletion	0.006
Contiguous Repeated Substitution	0.0009
Quadruple Contiguous Deletion	0.0001

b) Reducing the number of instructions required to express consecutive mutations: After statistically analyzing the most common operations in the alignments of the dataset, we defined a set of additional operations to describe contiguous mutations through a single operation (see Table 5-3). We complemented the 4 operations already considered (substitutions, deletion, insertion and Ninsertion) with four proposed contiguous operations based on the two most common biological mutations (substitutions and deletions). The four "contiguous mutations" most likely to happen were: Double, Triple and Quadruple Contiguous deletions, and the Contiguous Repeated Substitutions (substitutions in consecutive positions with same target base). Additionally, based on those probabilities (Table 5-3) we assigned the most efficient binary representation to the most common operations.

c) Preferring operations that required fewer bits: We defined categories of operations according to the gain in storage saving, in order to skew the specialized alignment and get optimal results (see Table 5-4). If there are several alignments with the same amount of mutations for a read, the one using less bits is chosen.

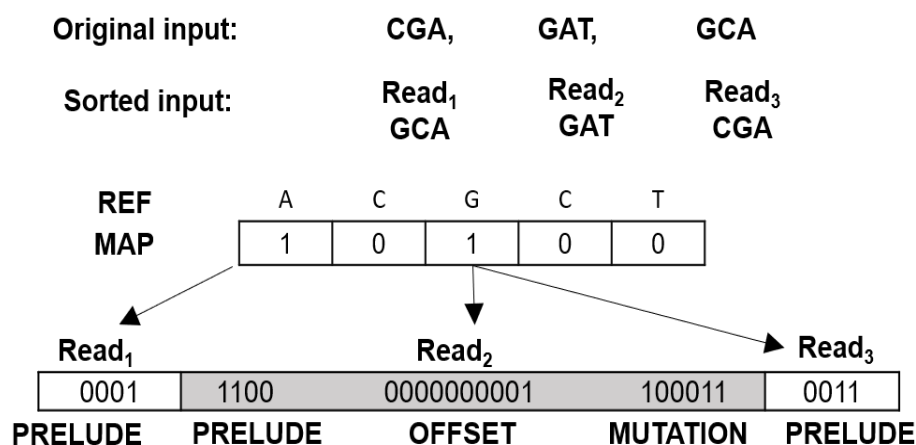
### 5.1.3.3. An example of instruction encoding

In Figure 5-4, we present an example of representing three reads using the encoding model. At the top of the figure we observe the original input reads, and below we see them sorted according to the mapping position. The map has 1's in positions one and three because those are the only locations where reads map. The first mapping position correspond to *Read<sub>1</sub>*, represented in the first

**Table 5-4.** Penalty categories

Cat	Operation	Storage Saving
1	Single Insertion, Single Substitution	Each mutation is stored using 16 bits
2	Single Deletion, Ninsertion	Allow storing 1 mutation using 14 bits
3	Contiguous Repeated Substitution	Allows storing 2 mutations using 16 bits
4	Contiguous Deletion	Allows storing 2 mutations using 14 bits
5	Triple Contiguous Deletion	Allows storing 3 mutations using 14 bits
6	Quadruple Contiguous Deletion	Allows storing 4 mutations using 14 bits

field: in this PRELUDE we see that this is the only read mapping in this position (0XXX) followed by the code describing an exact reverse matching (001); as the matching is exact the instruction is completely described through the PRELUDE. The next three fields correspond to *Read<sub>2</sub>*: The map says it matches in position three. The PRELUDE says the next read will map in this same position (1XXX), and this read matches approximately in forward mode (100). The following position is the OFFSET: 0000000001 because the mutation is in the second position of the read. In the MUTATION field the bits express: this is the last mutation in this read (1), and the operation (Oper) is a single substitution (000) of distance 5 (4+1, from C to A). Finally, the next field is for *Read<sub>3</sub>*: In the PRELUDE we see that no more reads map in this position and that it matches exactly through a complement transformation (011).

**Figure 5-4.** Example of instruction encoding to represent the three reads shown.

#### 5.1.3.4. UdeACompress implementation

UdeACompress was implemented in ANSI C. The map, which could require from tens of thousands to hundreds of millions of bits, is stored as an array of 64 bits integers in order to minimize memory accesses. For the instructions, we use instead an array of 8 bit integers in order to provide a

finer granularity that facilitates future parallel versions of the algorithm. For this reason, we use the 8 bits of the OFFSET field to store the 8 least significant bits (the offset suffix) of the whole offset. The 2 most significant bits from the third field are used to store the two most significant bits of the OFFSET (the offset prefix). This division lies in the fact that most of the times the bits of the offset prefix will be zero so there is no need to use them. This also brings a more uniform bit distribution.

The exact cost of representing an instruction will depend on the implementation approach and the hardware restrictions. Common hardware forces fixed size definition for data types, but specific hardware could allow particular sizes for user defined types, with a great impact in the cost of storage of an instruction.

If the encoder is implemented in a fixed data type size environment, grouping together all the preludes in the same array allows for storing two preludes in a single byte, and for every OFFSET-MUTATION pair 16 bits are needed. The implemented strategies were oriented to require the minimal bits to store each mutation, but in this current implementation saving less than eight bits in the encoding does not result in a direct reduction of the storage space. However, the resulting padding zeros will benefit the low level compression.

#### 5.1.4. Low Level Compression

The low level compression block is meant to compress two data streams that still can be reduced through a low level compression: (1) A small percentage of raw reads that did not match the reference, and (2) the binary representation for the map and the encoded instructions.

For the unmapped reads, Bzip2 was the clear choice, since it was the best performing general purpose tool according to our previous thorough study [24]. For the compression of the binary streams we considered the aforementioned traditional compressors (in section 3.1.1.1) and additionally some other relevant works: Paq806, Paq9, fpaqC, fpaq, flzp<sup>2</sup>, szip<sup>3</sup> and plzip<sup>4</sup>. The different algorithms make specific statistical assumptions about the data they would compress best. E.g. Huffman/arithmetic assume non-uniform symbol distribution, Lempel-Ziv assumes there would be repeated sequences, Run Length Encoding expects to find symbol repetitions, among others. Since we did not have much information, except that the binary data had a non-uniform symbol distribution we decided to benchmark their performance over the dataset. Again, bzip2 showed the best balance between efficiency and compression capabilities.

Both aforementioned problems are totally different, but we concluded the same approach suited

---

<sup>2</sup><http://mattmahoney.net/dc/>

<sup>3</sup><http://www.compressconsult.com/szip/>

<sup>4</sup><https://www.nongnu.org/lzip/plzip.html>

well for both cases. We used the low level interface of the library libbzip2 which is the current bzip2 API. More details about the implementation of bzip2 were presented in section 3.1.1.1 of this thesis.

## 5.2. Performance Evaluation

In this section we present the results of multiple performance tests applied to the implementation of UdeACompress and its integration with other modules of the workflow. Also, a comparison between UdeACompress and the most relevant specialized compressors in the state of the art is performed. In this comparison we included some referential compressors that were presented during the development of this research. Due to the requirements of an appropriate reference for such referential approaches, in these tests we used a publicly available dataset with the corresponding references. These results were published in the journal *Bioinformatics and Biology Insights* [170]

Tests were performed on a server with two Intel(R) Xeon(R) CPU E5-2620, 2.10GHz, 15360 KB cache, for a total of 12 cores and 24 threads, 40 GB of RAM in a shared memory architecture, a 1 TB SATA disk at 7200 RPM, and using Centos 7 OS (64 bits). Even when this version of UdeACompress was implemented sequentially, we let the third party software in this comparison to run in multi-thread mode.

Along with the results, we analyze the performance and efficiency of our proposal, taking into account advantages and drawbacks of the use of using UdeACompress for the compression of FASTQ files and read sequences alone.

Finally, as our compressor was developed to compress long read sequences, which are not very common currently but are expected to become a trend in the near future, we also performed tests using simulated data in order to measure the compression ratio under more favourable conditions.

### 5.2.1. Real Datasets Tests

We selected six FASTQ files: three plants and three bacteria; to compare the proposed algorithms in terms of compression ratio and speed against the best specialized compressors in the state of the art. Details about each dataset can be found in Table 5-5. This dataset was chosen in order to have a variety of: species, amount of reads, reads length (Illumina style) and reference file size. We used the reference indicated for each FASTQ as provided by the Sequence Read Archive <sup>5</sup>.

---

<sup>5</sup><https://www.ncbi.nlm.nih.gov/sra>

In some of the files the ID field was originally replicated in the comment field of each read. Then, we removed it to achieve a more accurate compression ratio results.

**Table 5-5.** Dataset description

Dataset	File Size (MB)	No of Reads	Read Length	Organism
<i>SRR1282409</i>	19119.61	57572520	151	Manihot esculenta (Plant)
<i>SRR3141946</i>	15755.74	67066956	100	Marchantia polymorpha (Plant)
<i>DRR000604</i>	14837.45	51732064	110	Oryza officinalis (Plant)
<i>SRR892505</i>	7040.81	21466082	150	Oxalobacteraceae bacterium
<i>SRR892403</i>	6668.85	28606666	100	Firmicutes bacterium
<i>SRR892407</i>	6104.16	18619528	150	Chitinophagaceae bacterium

### 5.2.1.1. Tested software

After reviewing tens of algorithms and tools for FASTQ compression and read sequences, we chose the latest versions of the most prominent software in the state of the art. In Table 5-6 we summarize relevant information about each program: the approach used for the compression, the target data and the number of threads used by default. All programs were configured in lossless compression mode and the remaining configuration parameters were left at their default values. Even though Quip and FASTQZ allowed enabling a referential compression mode, in our previous tests it was evident that such option did not improve significantly the compression ratio while increasing the execution time considerably; for that reason we discarded such configuration. In the following experiments we report the minimum of the execution time of three replicas performed.

All the tested programs, their approach, and the configuration of their default multi-threaded executions shown in Table 5-6.

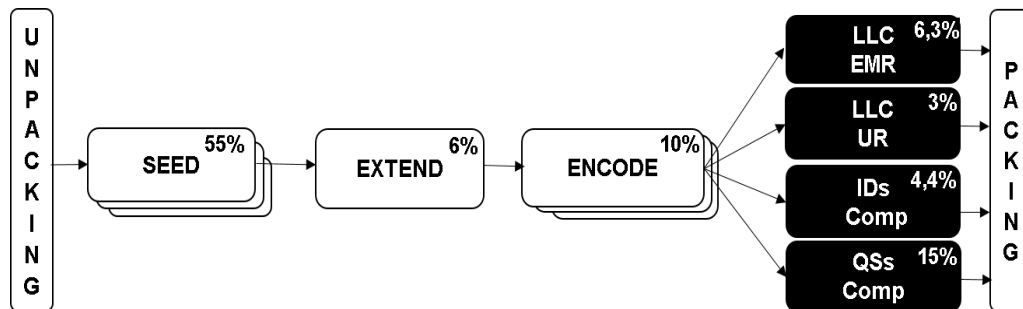
The only program in the literature that is directly comparable to UdeACompress is LWFQZIP2, since it performs an alignment-based referential compression of FASTQ. It is therefore, the only possible fair comparison of quantitative results. We present comparisons with the other approaches, as mere reference points and in order to provide a wider perspective for the reader. Additionally, we must make clear that ORCOM is not directly comparable to any of the tested programs since it is a reference free compressor designed for large collections of FASTQ files, instead of compressing a single FASTQ file. However, given its relevance in the state of the art and its impressive performance we decided to include it here. In the case of Kpath, which generates a file to preserve the reads original order, this size was excluded of the reported compression ratio. We experienced some problems executing Assembltrie, which compressed only two of the six files in the dataset.

**Table 5-6.** Compression software description

Tool	Approach	Data Object	Used threads
DSRC 2.0 [51]	Non-referential	FASTQ	Maximum available
QUIP [22]	Non-referential	FASTQ	1-2
SCALCE (+PIGZ) [29]	Non-referential	FASTQ	4
FASTQZ [50]	Non-referential	FASTQ	3-4
LEON [69]	Referential	FASTQ	Maximum available
LWFQZIP2 [75]	Referential	FASTQ	10
KPATH [70]	Referential	Read sequences	10
ORCOM [74]	Non-referential	Read sequences	8
HARC [171]	Non-referential	Read sequences	8
Assembltrie [172]	Non-referential	Read sequences	8

### 5.2.1.2. Parallelism estimation

Since the current implementation of UdeACompress runs in single thread mode, we developed a simple parallel model that allows us to estimate more comparable speed metrics with respect to the other tools that all support multi-thread execution. In such a model, we only consider the straightforward parallelization of the most compute intensive tasks that are known to be parallelizable. This analysis would provide the necessary information to prioritize the optimizations to be performed in the next step of this research (Chapter 6).



**Figure 5-5.** Profile of the sequential version of UdeACompress. Boxes show the percentage of time consumed by each function. Black boxes correspond to third-party software. LLC EMR: is the low level compression of the encoded mapped reads, LLC UR: is low level compression of unmapped reads, IDs Comp: Identifiers compression, and QSSs Comp: Quality scores compression.

Figure 5-5 shows a simplified block diagram of UdeACompress with profiling information. The profiling data was obtained as an average of testing all the FASTQ files in the dataset. We used *gettimeofday()* which has microsecond precision, observing no significant variation among the replica tests. Since there are only memory and disk operations in the packing/unpacking steps, we did not



not include such functions in this profiling. The decompression task was not considered because the structure of the array of encoded instructions demands that decompression has to be performed sequentially.

There were three blocks consuming 80% of the total time of UdeACompress: 1) the seed calculation performed during the alignment, 2) the compression of QSs and, 3) the encoding of the instructions. Since QSs compression was performed by a third-party software, we did not consider it as an immediate choice for parallelization. Equation 5-1 presents an analytical model to estimate the speedup resulting from parallelizing the seed and the encode processes. In the following figures of performance, the blue dashed line bar (named UdeACompressP) right next to the UdeACompress blue bar, represents this estimation.

$$T_{UdeACP} = \frac{T_S}{N} + T_{Ext} + \frac{T_{Enc}}{N} + \max(T_{Id}, T_{Qs}, T_{LLCE}, T_{LLCU}) \quad (5-1)$$

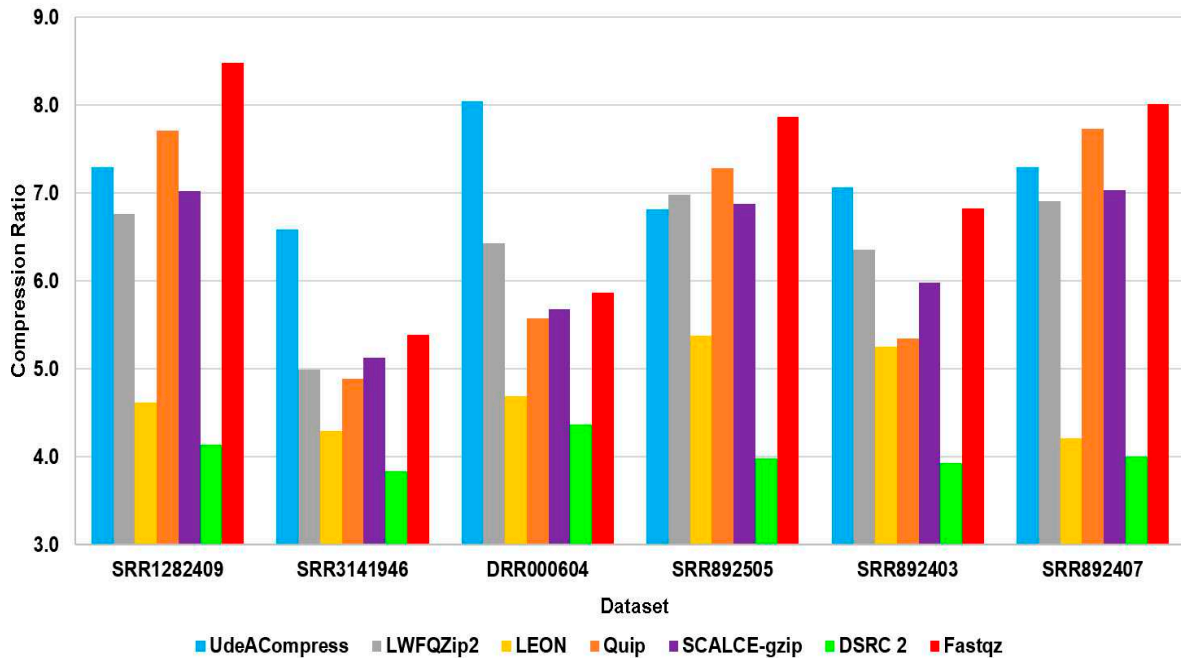
$T_{UdeACP}$  is the estimated time of the parallel implementation of UdeACompress during compression,  $T_S$  is the time corresponding to the sequential execution of the seed phase performed during alignment,  $N$  is the number of available threads in the architecture (24),  $T_{Ext}$  is the time corresponding to the sequential execution of the extend phase performed during alignment,  $T_{Enc}$  is the time spent in encoding sequentially the alignment instructions,  $T_{Id}$  is the time spent compressing the IDs,  $T_{Qs}$  is the time spent compressing the QSs,  $T_{LLCE}$  is the time consumed by the low level compression of the encoded mapped reads and  $T_{LLCU}$  is the time consumed by the low level compression of the unmapped reads. The last four steps can be performed in parallel since there is no dependency among them.

### 5.2.1.3. Compression ratio

One of the main goals of this work is to measure and compare the compression capabilities of UdeACompress. In Figure 5-6) we show the compression ratio, that is, the ratio between the FASTQ file size and the compressed file size, achieved for the six FASTQ files in the dataset.

Results show UdeACompress achieves similar or better compression ratios than the best state of the art programs, with an improvement between 4% and 27% respect to the second best program for three of the input file tests. In the rest of the datasets the maximum difference between UdeACompress and the highest compression ratio is only 14%. In five of the six datasets we achieved higher compression ratios than the rest of referential FASTQ compressors, and in the case of the exception the best compressor is only 2% above.

Figure 5-7 shows the compression ratio corresponding to the read sequences only, without taking

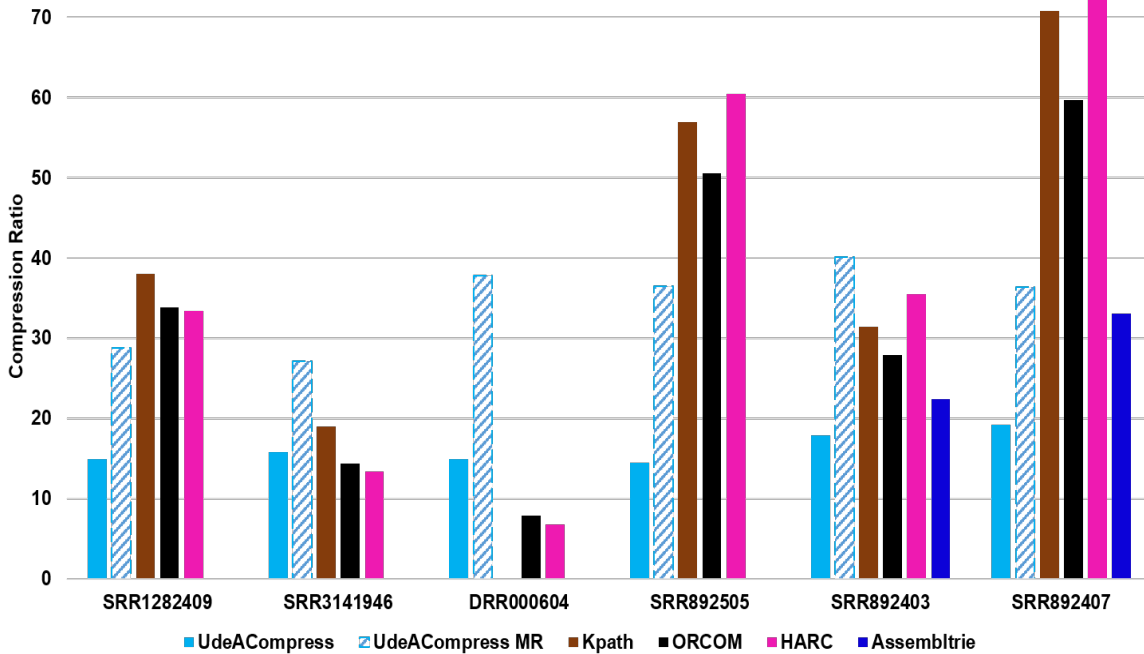


**Figure 5-6.** Compression ratio for FASTQ files.

into account the IDs and Qs. This experiment allows us to evaluate in more detail the capabilities of the algorithm we developed, given that IDs and Qs are compressed using third-party software. In such tests, UdeACompress achieved a high compression ratio for two of the largest inputs, for the rest of cases the performance was lower. Exploring the files content, we found that the main reason for the good performance of UdeACompress for the third input is that it contains a higher amount of consecutive N's, which is effectively harnessed by the encoding scheme.

The inclined lines bar in Figure 5-7 shows the compression ratio of UdeACompress when processing mapped reads only, which naturally reflects a much better performance of our method in such scenario. Our compressor is significantly affected by the compression of unmapped reads, which is done using a general purpose compressor (bzip2). The negative impact becomes evident when comparing both UdeACompress bars. It decreased the compression ratio (in average) up to 50%, respect to the compression of mapped reads only. The other programs in this experiments use methods that are not affected by the phenomenon of the so called unmapped reads. The efficient compression of unmapped reads requires a very different approach that is to be included in future versions of UdeACompress.

- The amount of unmapped reads,
- Read lengths, since at least two bits in the offset are permanently underused in each mutation when reads are short,
- The amount of mutations per read,



**Figure 5-7.** Compression ratio for the read sequences. The bar filled with inclined lines (UdeACompress MR) represents the compression ratio of UdeACompress discarding the unmapped reads.

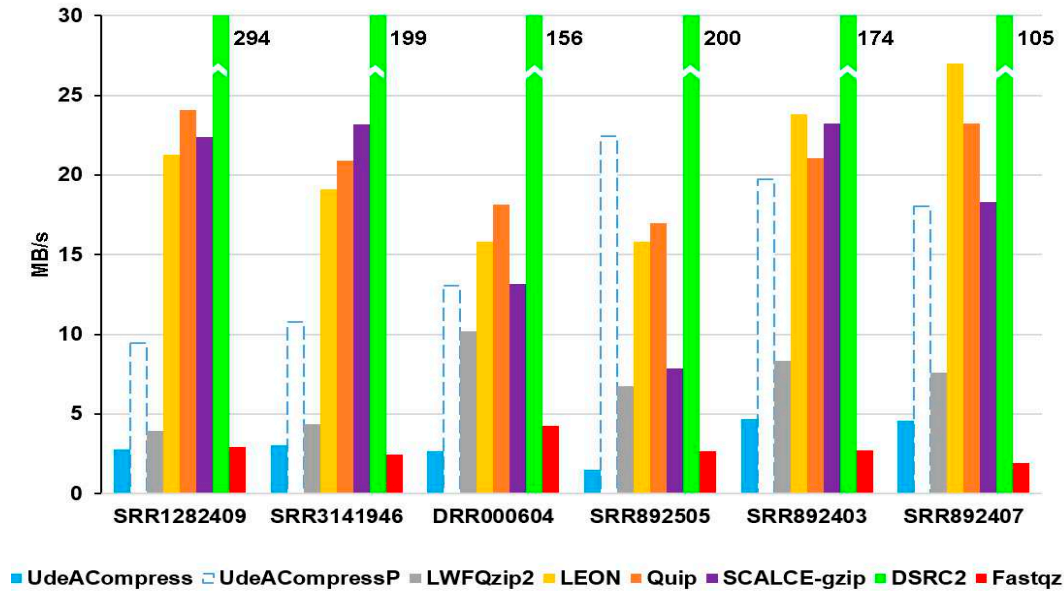
- The fixed sizes for the software data types which limits the benefits of bit encodings that do not match the established sizes and,
- The referential method used, as it is, does not fully harness possible inter-reads redundancies since it focuses on reads-to-reference encoding.

It must be noticed that the performance of UdeACompress was always above the performance of all referential compressors for FASTQ files.

#### 5.2.1.4. Throughput during compression and decompression

It is expected that high compression ratios and fast performance are conflicting goals. Even though we were focused on compression ratio, we also wanted to compare the execution time considering that it is a very important usability factor. We present the following results in terms of throughput, defined as the amount of Megabytes per second (MB/s) processed for each program during the compression and decompression.

Figure 5-8 shows throughput during compression of the FASTQ files. Although the sequential version is outperformed by most of the others, the parallel estimation tell us UdeACompress throughput would be similar to most of the fastest programs available. Our algorithm is sensitive not only



**Figure 5-8.** Throughput during compression of FASTQ files. The dashed line (UdeACompressP) represents the estimated throughput of a parallel implementation of UdeACompress using the 24 threads available in our setup.

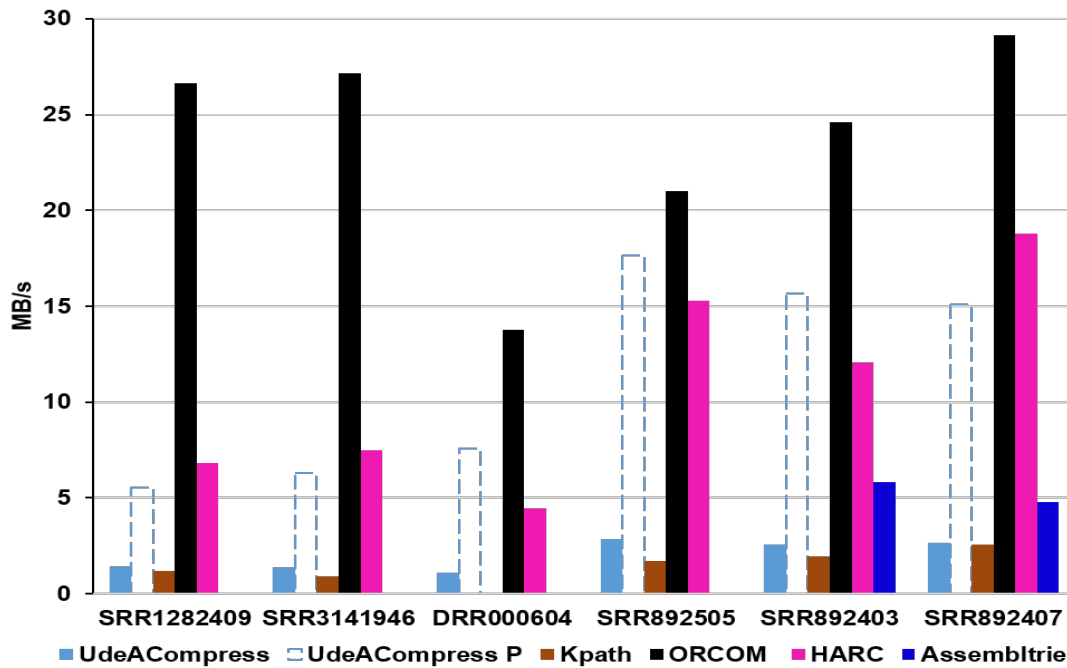
to the size of the input file (length and number of reads) but also to the size of the reference, increasing significantly the amount of memory and CPU required. On average, our sequential and parallel algorithms processes data at around 3-4 MB/s and 10-20 MB/s respectively.

Figure 5-9 shows the results of compressing the read sequences only. UdeACompress was faster than Kpath but slower than the other applications tested, at a variable rate. This is explained mainly by the fact that, as the input and the references increase their size, the alignment process takes longer and UdeACompress speed decreases significantly. Furthermore, the main goal of this version of UdeACompress was to achieve high compression ratios.

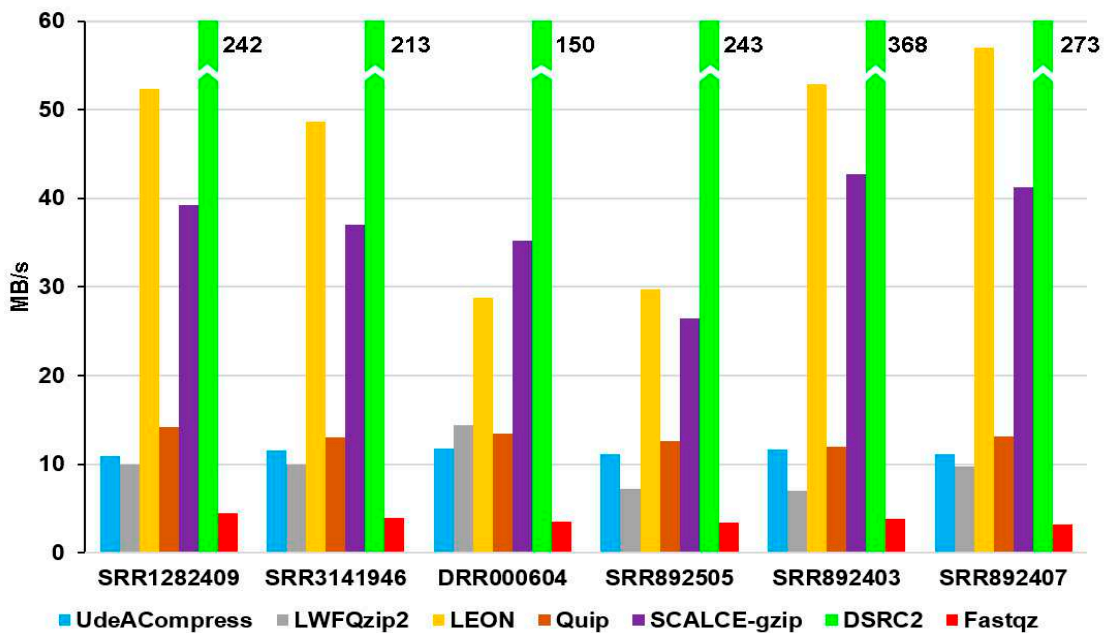
In the two compression scenarios presented, a parallel version put us in a competitive position with the fastest programs available. Limitations in the performance improvement of UdeACompressP in Figure 5-9, show the impact of the low level compression which is also performed sequentially since there is no parallel version for the bzip2 API yet.

Decompression results corresponding to all the streams inside the FASTQ file are presented in Figure 5-10. UdeACompress shows an average throughput of 11.5 MB/s, being faster than the other alignment-based compressor (LWFQZIP2).

Throughput during decompression of the read sequences only is in Figure 5-11. UdeACompress behaves consistently, with no big variation at an approximate rate of around 11 MB/s, in overall

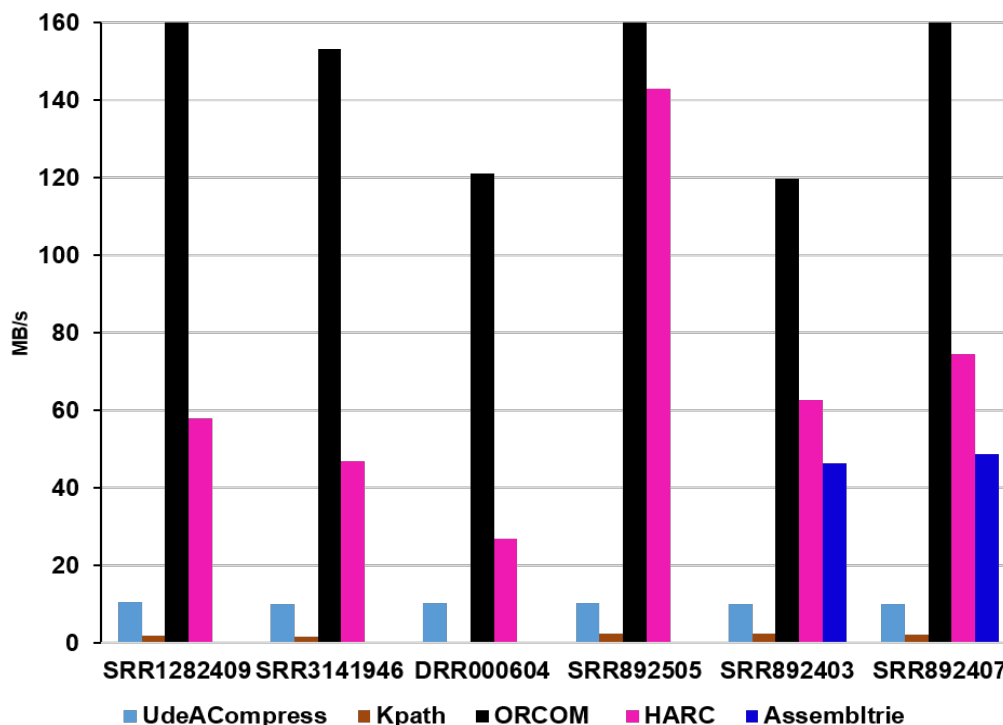


**Figure 5-9.** Throughput during compression of the read sequences. The dashed line (UdeACompressP) represents the estimated throughput of a parallel implementation of UdeACompress using the 24 threads available in our setup.



**Figure 5-10.** Throughput during decompression of FASTQ files.

5× faster than Kpath and below the other applications.



**Figure 5-11.** Throughput during decompression of the read sequences.

#### 5.2.1.5. Peak memory usage

Although not part of our main goals, we present here experiments to measure the memory consumption of UdeACompress and the other applications for the sake of completeness. Table 5-7 is presented as part of the analyzes commonly carried out in this field.

To measure the peak memory consumption (in MB), all programs were executed using their default configuration, both during compression and decompression.

In terms of memory usage, UdeACompress is not as thrifty as the other compressors for the whole FASTQ file. However, in the field of read sequences compression it is common to expect much higher memory requirements, due to the complexity of the techniques and data structures involved. Both during compression and decompression, UdeACompress could be classified among the most memory demanding tools, along with Kpath and Assembltrie

The amount of memory required by UdeACompress during compression is almost proportional to the input of the FASTQ file. In overall, during decompression UdeACompress demanded around

**Table 5-7.** Peak memory consumption during compression and decompression (MB). Below each dataset we show the total size of the full FASTQ input file and the size of the read sequences only.

	<b>SRR1282409</b>		<b>SRR3141946</b>		<b>DRR000604</b>		<b>SRR892505</b>		<b>SRR892403</b>		<b>SRR892407</b>	
FASTQ	(19119 MB)		(15755 MB)		(14837 MB)		(7040 MB)		(6668 MB)		(6104 MB)	
Reads only	(8346 MB)		(6459 MB)		(5477 MB)		(3092 MB)		(2756 MB)		(2682 MB)	
	<b>Com</b>	<b>Dec</b>	<b>Com</b>	<b>Dec</b>	<b>Com</b>	<b>Dec</b>	<b>Com</b>	<b>Dec</b>	<b>Com</b>	<b>Dec</b>	<b>Com</b>	<b>Dec</b>
<b>FASTQ Compressors</b>												
<b>Non-referential</b>												
<b>Quip</b>	384	383	385	383	392	391	384	369	384	369	384	369
<b>SCALCE</b>	5226	1036	5242	1038	5319	1037	5220	1036	5220	1036	5220	1036
<b>DSRC 2.0</b>	229	1107	239	1104	226	1028	238	1122	225	1128	237	969
<b>Fastqz</b>	1528	1528	1528	1528	1528	1528	1460	1460	1460	1460	1460	1460
<b>Referential</b>												
<b>UdeAC</b>	10639	9691	7578	7030	7162	7098	3449	3680	3414	3791	3328	3419
<b>LWFQZIP2</b>	1847	1843	1835	1835	1956	1956	662	662	678	678	1729	1728
<b>Leon</b>	5625	2839	5361	2875	5191	2895	5654	2713	4901	2197	5673	2713
<b>Read sequences compressors</b>												
<b>Kpath</b>	30886	14975	25412	12981	-	-	11345	6611	13058	9443	9830	5745
<b>ORCOM</b>	9416	2345	9365	2312	7296	1631	6573	1345	2440	639	5871	1631
<b>UdeAC</b>	10639	9691	7578	7030	7162	7098	3449	3680	3414	3791	3328	3419
<b>HARC</b>	2890	1257	3269	1465	2554	2547	1159	95	1378	224	2985	99
<b>Assembltrie</b>	-	-	-	-	-	-	-	-	16201	4193	9286	3629

25% less memory. One of the issues to be tackled in future versions of UdeACompress is finding efficient ways of handling the data structures related to the alignment and the reads encoding, where the most memory is consumed. Particularly, the biggest data structures are required during the seeding and extend phase of the alignment, and in the process of encoding the alignment data to generate the binary instructions. Despite exhibiting high memory usage compared to other tools, the absolute values measured are within the memory capacity of high performance machines commonly available to bioinformatics research centers.

## 5.2.2. Simulated Datasets Tests

Since UdeACompress encoding scheme is designed to be more efficient when handling longer reads than what is commonly found today in public databases (a few hundred bases), we decided to analyze its compression potential over inputs with longer reads and varying the amount of mutations per read. Since these experiments are only related to changes in the read sequences properties, we only report the compression ratio of the read sequences, and we only compare to the applications compressing such stream only.

Subsequently, a set of simulated data files was created to test the performance of UdeACompress under different scenarios. The goal of these experiments was to test compression capabilities of

UdeACompress measuring: (1) the effect of the read lengths variation, since this was the main factor considered in the instruction design, (2) the effect of the number of mutations per read, as the instruction sizes grow when the number of mutations increases; and (3) The effect of the redundancy among different reads, expressed as the coverage parameter.

For such dataset, we built files with up to 6 GB of read sequences only; generated from a human reference in order to expand the range of species included in this evaluation.

### 5.2.2.1. Experimental setup

We built a tool using ANSI C to create simulated FASTQ files in Illumina style, considering a set of relevant parameters: read length and maximum number of mutations per read (see Table 5-8). Also, we defined probability functions to calculate the matchings, mutations, bases, offsets and to calculate the maximum number of mutations per read. Outputs include the required IDs and QSs, but we used empirical values for the generation of such fields since they are meant to be discarded by all the compressors in these tests. To make a fairer comparison, only classical matchings (forward and reverse) were used to generate the simulated data files and not skew the results to our benefit. All the datasets were built using a human reference, the Homo sapiens chromosome 6, GRCh38.p12, primary assembly <sup>6</sup>, downloaded on June 14, 2018.

**Table 5-8.** Simulated tests configuration

Parameter	Variation
Reference	Homo sapiens GRCh38, chromosome 6
Read Length	128, 256, 512, 1024; default 1024
Amount of reads	6000000
Coverage	[25×, 250×]; default 70×
Maximum percentage of mutations per read	[0%, 10%], 25% ; default: 10%
Number of mutations distribution	Exponential
Offsets distribution	Uniform
Mutation probabilities	According to Table 5-3
Matching probabilities	Uniform for: Forward and Reverse
Base probabilities (substitutions)	Uniform for: A, C, T, G; for N=0.08
Base probabilities (Insertions)	Uniform for: A, C, T, G

Mutations were adjusted to an exponential distribution [42]. After a previous statistical analysis of FASTQ files, we found that a typical upper bound for the quantity of mutations per read is below 10% of the read size, so we generated a maximum number of mutations between 0% and approximately 10% of the read length; to test the performance of UdeACompress in situations of

<sup>6</sup>[https://www.ncbi.nlm.nih.gov/nuccore/NC\\_000006.12?report=fasta](https://www.ncbi.nlm.nih.gov/nuccore/NC_000006.12?report=fasta)



non-optimal alignment between the reads and the reference. Probabilities of occurrences for bases and mutations were established according to the values presented in Table 5-3.

By default, the maximum of number of mutations per read was set to 10% of the read length; reads length was set by default to 1024 bases, and the calculated coverage of the simulated data was approximately 70 $\times$ , according to recommendations <sup>7</sup> [173]. Unmapped reads were not considered in these tests since their processing corresponds to a third-party software.

Although, we intended to test the same compressors for read sequences presented in section Compression ratio, preliminary tests revealed that ORCOM was not able to compress any of the files with large reads. On the other hand, the documentation of HARC and Assembltrie states they cannot handle large reads either. Only Kpath matched the requirements to compress the simulated data, which highlights the contribution of our approach.

#### 5.2.2.2. Read lengths effect

The first factor to impact UdeACompress performance is the read length. Its effect on the compression ratio was measured while varying the maximum number of mutations per read, as shown in Table reftab:ArtTConf. The maximum percentage of mutations was restricted considering only 10% and 25% mutations per read. Results are presented in Figure 5-13.

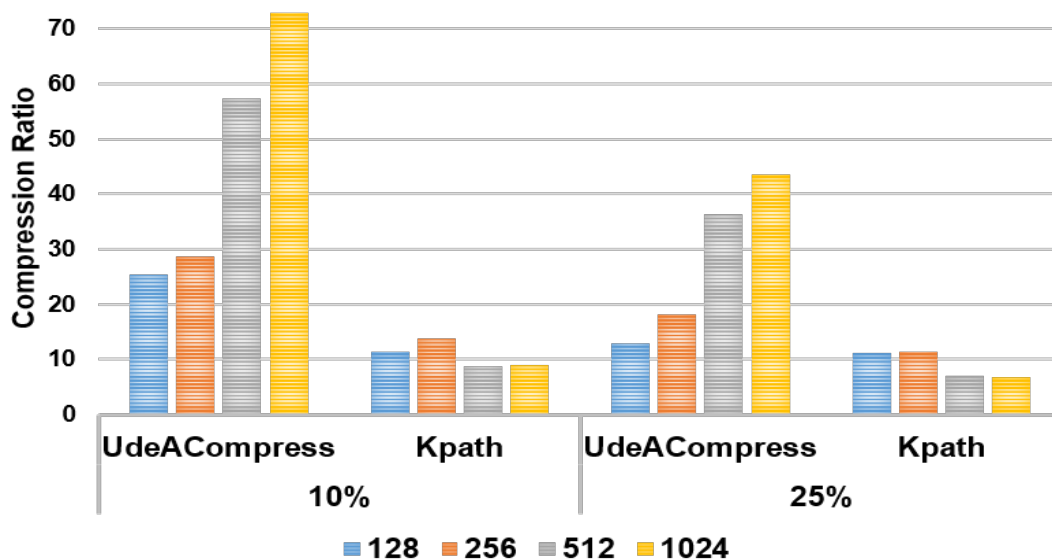
The compression capability of UdeACompress significantly increases as the reads get longer and the percentage of mutations decreases. This is expected since with longer reads the instructions encode more information using the same amount of bits, and the impact of storing the map is reduced because the same structure is used to represent more data.

Figure 5-13 also shows a scenario where shorter reads (128, 256) have a very high percentage of mutations (25%), which impacts negatively our referential compression. Even in such cases, the performance UdeACompress is still acceptable compared to the other evaluated application. Considering a significantly high maximum percentage of mutations (as 25%) and reads of length equal or superior to 512, UdeACompress achieved a compression ratio between 36 $\times$  and 44 $\times$ , while all but one of the applications in the state of the art cannot even process reads of such length.

#### 5.2.2.3. Effect of the maximum number of mutations

For this test, we generated reads with a maximum amount of mutations between 0% and 25%. It must be noticed that, because of the exponential distribution, the percentage of reads with exact matchings (zero mutations) is always higher than any other number of mutations probability. Also, since contiguous mutations are represented as single mutations in the instruction design, the

<sup>7</sup><https://www.illumina.com/science/education/sequencing-coverage.html>



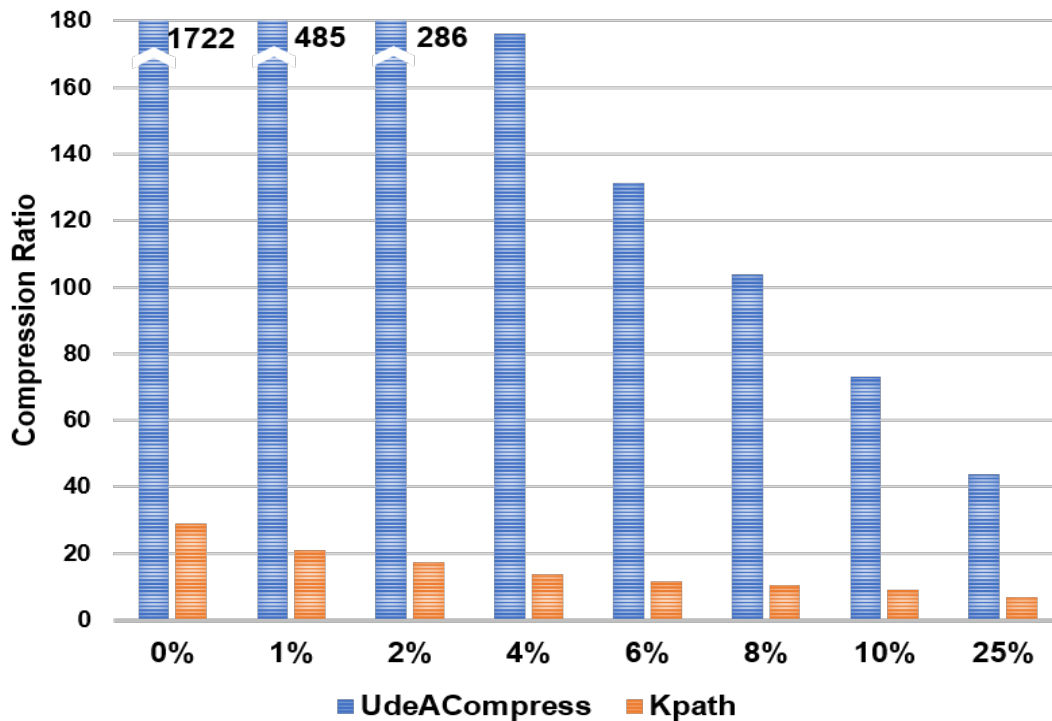
**Figure 5-12.** Compression ratio of the read sequences through the variation of the read length and the percentage of mutations per read.

exact amount of mutations in each read could be a few mutations more than what the percentage expresses. Figure 5-13 shows the behavior of UdeAcompress compression ratio as the maximum amount of mutations per read is increased.

As expected, the best performance is achieved with fewer mutations per read. The difference between both compressors tends to decrease as the number of mutations is increased. A very small percentage of mutations [0% - 2%] could be considered unrealistic in practice, since it refers to almost exact alignments which are not likely to happen when handling reads of length 1024. But, even in the very unfavourable scenario of a maximum 25% of mutations per read, UdeACompress compresses over  $6\times$  more than Kpath. Finally, it can be noticed that the range of the compression ratio for a  $70\times$  coverage and typical percentages of mutations (8% to 10%), can be estimated between  $70\times$  and  $100\times$ .

In a scenario that included unmapped reads, UdeACompress would still compress better than the rest of available software.

When sequencing technologies for longer reads arrive, it must be studied the real impact that longer reads could have in the amount distribution of mutations.



**Figure 5-13.** Compression ratio of the read sequences through the variation of the maximum percentage of mutations per read.

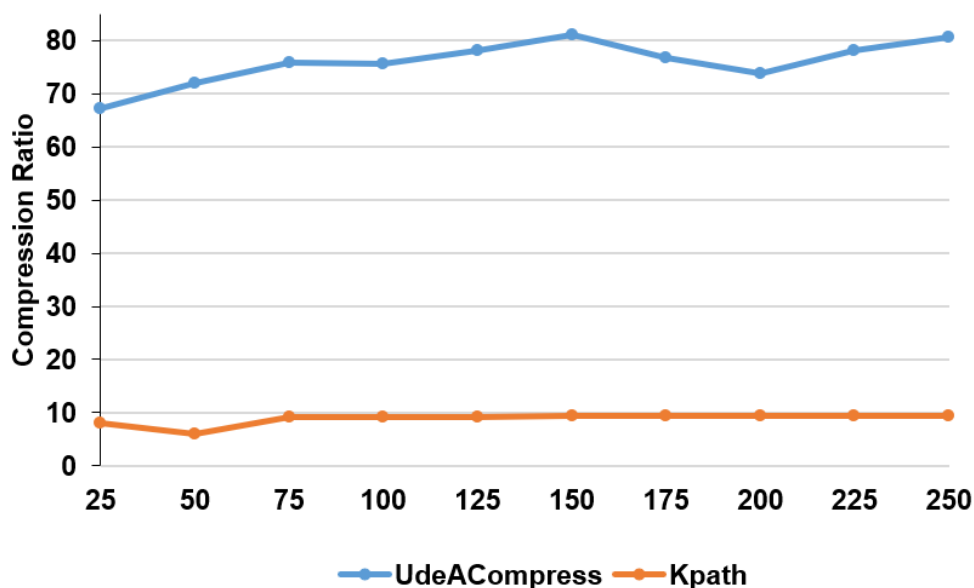
#### 5.2.2.4. Effect of coverage

Coverage ( $C$ ) is an important measure of redundancy usually considered in the case of the compression of genomic data. It was estimated using equation 5-2<sup>8</sup>:

$$C = \frac{L * N}{G} \quad (5-2)$$

where  $L$  is the length of the reads,  $N$  is the number of reads in the file and  $G$  is the reference length. Figure 5-14 shows the results of experimenting with different coverage values for UdeACompress and Kpath. A slight increase in compression ratio can be observed for UdeACompress as coverage grows due to the increased redundancy. UdeACompress' limitation to harness all the available redundancy among reads is evident as the compression ratio increases less than 15% (from 67× to 81×), while the coverage has increased 10×. This fact is the consequence of an approach that focuses on exploiting the read-to-reference redundancy rather than the similarities among reads (read-to-read). The small gain observed in Figure 5-14 is a consequence of the shared map in our design, which allows to take a little advantage of the existing redundancy among the reads. Applying strategies in the style of delta encoding over the encoder output, and before the low level compression, could help improving the final compression ratio.

<sup>8</sup>[https://www.illumina.com/documents/products/technotes/technote\\_coverage\\_calculation.pdf](https://www.illumina.com/documents/products/technotes/technote_coverage_calculation.pdf)



**Figure 5-14.** Compression ratio of the read sequences with varying coverage values.

Nevertheless, UdeACompress performance was consistently above Kpath, exhibiting a compression ratio at least  $6\times$  higher. On the other hand, Kpath also showed limitations to exploit the redundancy in the reads introduced by the increased coverage, exhibiting a constant behavior.

### 5.3. Summary

In this chapter we have presented the implementation of UdeACompress, a FASTQ read compressor and the core of the multi-technique compression block of the workflow for FASTQ referential compression presented in the previous chapter.

UdeACompress consists of a set of internal blocks responsible for performing the main tasks: specialized alignment between the reads and the reference, sorting the reads according to their mapping position, the encoding of the alignment data through a map and a set of instructions, and the compression of such encoded information using compression algorithms based on the BWT transform. Some blocks of third party software were used to test it, allowing to perform comprehensive experiments of FASTQ and read sequences compression.

Several performance metrics related to compression capabilities and execution speed were measured during tests, to evaluate and compare our proposal against the top specialized compressors in the state of the art. When compared to the other alignment-based referential compressor avail-

able, UdeACompress had similar or better compression ratios, producing files 14% smaller and decompressing  $1.3\times$  faster, on average. During compression, throughput was shown to be similar to most other programs when including the parallel estimation, and lower than the average for the sequential case. During decompression the running times were always in the average range. The additional mutations and matchings introduced in UdeACompress seemed to have a positive impact in the compression, but this still needs to be studied as well using multivariate statistics. In summary, results have shown that UdeACompress performance is competitive when compared to the most relevant tools in the state of the art for FASTQ compression, at the expense of extra memory consumption.

As UdeACompress was envisioned thinking in the future scenario of longer reads, simulated data was generated to measure the real capabilities of UdeACompress when compressing reads sequences only, exploring deeply the effect of the multiple variables involved. The compression ratio of UdeACompress was significantly increased when processing longer reads, being always above its only competitor, since most of the state of the art software is not ready to handle long reads. A decline in the compression ratio was observed as the maximum number of mutations increased, but in typical cases this value was around  $60\times$ . However, experiments showed that the proposed compressor can still be improved, to achieve a better harnessing of the redundancy in the input data. In future versions, it should be explored the results of combining the inner low level compressor with strategies as delta encoding or Markovian models that could exploit better the similarities among the read sequences.

The algorithm runtime is sensitive not only to the length and number of reads in the FASTQ input file, but also to the size of the reference, increasing the amount of CPU and memory required, especially for the alignment. Even though the execution times of UdeACompress were acceptable in comparison to relevant software in the state of the art, the estimation model allowed us to predict a noticeable superior performance if the seeding and encoding blocks of UdeACompress were parallelized.

Compression of QSs and unmapped reads had a great impact in the compression ratio of a FASTQ file so specialized strategies for their compression need to be developed for this task. Also, different strategies for compressing the IDs must be tested, since the re-arrangement of the reads performed by UdeACompress could handicap the delta encoding that was used to compress them.

Finally, UdeACompress stands out as an effective alternative for compressing not only a FASTQ file, but also the genomic alignment data. However, after evaluating the results the obvious priority was to accelerate the main bottlenecks found in the profiling of UdeACompress, in order to make the runtime more competitive.

The work presented in this chapter has been partially published in:

**Guerra, A.**, Lotero, J., Aedo, J., and Isaza, S. (2019). Tackling the challenges of FASTQ referential compression. *Bioinformatics and Biology Insights*, volume 13, SAGE.

Lotero, J., Benavides, A., **Guerra, A.**, and Isaza, S. (2018). UdeAlignC: Fast Alignment for the Compression of DNA Reads. In *2018 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE.

## 6. Improving the Execution Speed Through Parallelism

Although running times of UdeACompress were acceptable compared to its competitors, the absolute runtime was still high. From the user's point of view, any performance improvement would have a positive effect. According to our previous profiling (in section 5.2.1.2), there was space for an important speed-up through harnessing the features of HPC hardware, which are commonly available in bioinformatics centers. In consequence, a parallelization strategy was applied to accelerate the main bottleneck in the proposed compressor.

The profile of UdeACompress showed the blocks most susceptible to be accelerated: The seeding phase of the aligner (55%) and the encoding stage (10%). Even the QS compression consumed a higher percentage of the runtime (15%) compared to the encoder time, this was not a feasible choice since this was done with third-party software.

Our efforts to accelerate UdeACompress, focused on harnessing the SIMD capabilities of today's high-end processors. Furthermore, we also explored the use of multi-threading to exploit multicore architectures. This chapter describes the basic concepts, the methodology, and the results along with the corresponding scalability analysis.

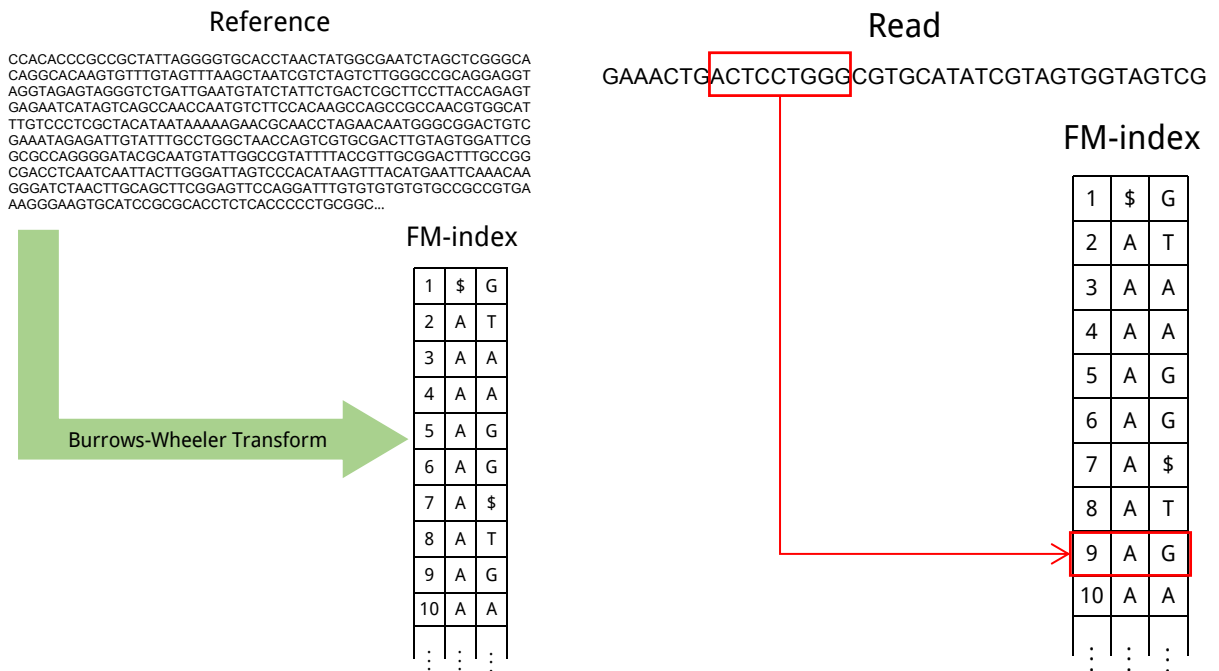
### 6.1. Accelerating the Alignment Algorithm

The alignment of reads to a reference genome allows the compressor to know the region in the reference genome where a particular read best matches. Once such a place is found, the compressor can proceed to encode the differences. Because of the large sizes of genomes and the amount of reads to be compressed, the alignment process in itself is highly demanding in terms of computing time. Hence, the implementation of our referential compressor can greatly benefit from using faster aligning algorithms.

The UdeAcompress aligner, presented in [163], employs the well known Seed and Extend strategy [174–176]. As shown in Figure 5-5 more than 50% of the running time of our compressor was spent in the seeding phase; in consequence, it became our acceleration goal.

### 6.1.1. Profiling the Seed Algorithm

Seeding consists of extracting a small substring or seed from the read and searching across the reference to find an exact coincidence; the algorithm admits no mismatches but allows fast searching. If such coincidence is found, the whole read is then aligned to the reported position in the reference at the extend phase, where an algorithm that supports mismatches, insertions and deletions is used. Figure 6-1 provides a visual outline of the the seeding process, explained as follows.



(a) A FM-index structure is created from the reference genome. This stage needs to be executed only one time, since results can be saved for future alignments. (b) A subsequence or seed is extracted from the read and sought within the index.

**Figure 6-1.** Stages of the seeding phase [163].

The seeding phase is implemented using an FM-index. FM-index algorithm generates a compressed representation of the Burrows-Wheeler transform (BWT) [28] from the reference in a preliminary step, as a way to create an index to quickly look for short seed sequences [177] using the first and last column of the BWT. The Burrows-Wheeler transform is obtained after a lexicographical sorting of all possible permutations of all the bases in the reference, which takes a considerable amount of time [159]. However, this task needs to be executed only once per reference and stored for later uses. On the other hand, it must be calculated for each of the FASTQ files in the input. In a deeper profiling and analysis of the inner structure of the seeding algorithm we found out that more than 80% of the runtime was invested in building the FM-index, which will be presented next.



### 6.1.1.1. FM-Index

In 2000, Ferragina and Manzini proposed a data structure to combine the best of Burrows Wheeler Transformation and suffix arrays, called the FM-index (“Full-text index in Minute space”). Burrows Wheeler Transformation and suffix arrays will be explained in detail in the subsections 6.1.1.2 and 6.1.1.3.

FM-Index aims at achieving a space-efficient representation of the input data, to provide fast indexing and searching; complementing the Burrows-Wheeler Transform (BWT) representation with auxiliary data structures and emulating the functionality of a suffix array (SA) [178]. E.g. While the SA of a typical human genome (3 billion bases) needs 14 GB, the corresponding FM-index could take only 1.5 GB.

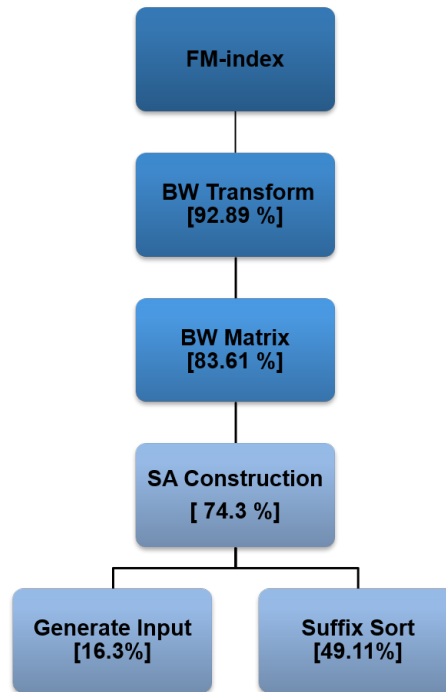
With an FM-Index, a reference sequence can be kept compressed, and there is no need to decompress the whole sequence for pattern searching. Patterns can be searched in a large reference by only decompressing a small part of it, which results in a significant reduction in storage and access costs [159]. This can be done through LF-mapping techniques and backward searching as shown by [177], using a defined set of operations (*find*, *locate*, *rank*, *access*, *extract*) which allow to query (over the BWT) in constant time and without the original text. The look-up in the generated index is completed in  $O(p)$  time, where  $p$  is the seed length. As it will be explained next, we focused our efforts in parallelizing the Suffix Arrays Construction inside the FM-Index, so we invite the reader to review the literature [159, 177] to get a detailed explanation of such processes. The construction of the FM-Index is computationally intensive, in a former profiling using gprof (as shown in Figure 6-2), more than 90 % of the cost of building an FM-Index was spent in calculating the BWT.

### 6.1.1.2. Burrows Wheeler Transform

The Burrows Wheeler algorithm [28] transforms a string  $S$  of  $N$  characters by forming the  $N$  rotations (cyclic shifts) of  $S$ , sorting them lexicographically, and extracting the last character of each of the rotations. A string  $L$  is formed from these characters, where the  $i$ th character of  $L$  is the last character of the  $i$ th sorted rotation.

In addition to  $L$ , the algorithm computes an index  $I$  of the original string  $S$  in the sorted list of rotations. Along with BWT, the authors also proposed an efficient algorithm to compute the original string  $S$  given only  $L$  and  $I$ . The BWT has been widely applied in DNA compression nowadays [179] [28].

In practice, the BWT output string results from building the Burrows Wheeler Matrix (BWM). The BWM is a matrix with all the rotations of the input stacked vertically, whose rows are sorted lexicographically. Figure 6-3 shows an example of calculating the BWM(T) for the input  $T = \text{”attcatg”}$ .



**Figure 6-2.** Succinct call-graph of the process of building the FM-Index.

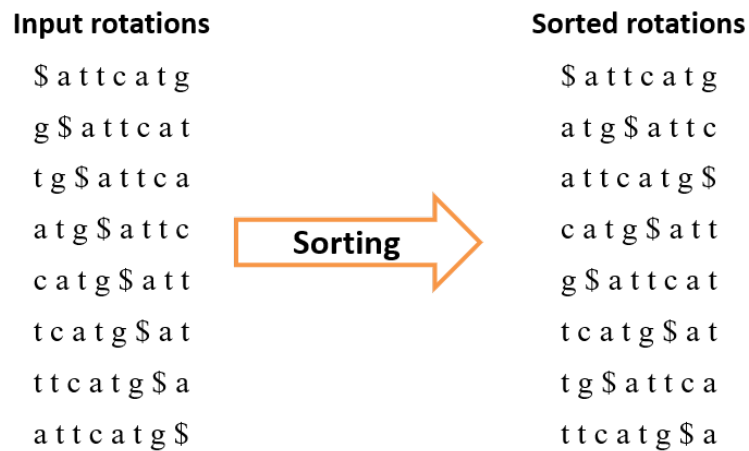
The last column of the BWM, read from top to bottom, is the resulting  $BWT(T)$ : gc\$tttaa. This procedure is very convenient since it is reversible and the transformed string can be compressed more efficiently than the original. The Burrows-Wheeler Matrix is constructed based on the SA of  $T$  ( $SA(T)$ ). Both processes involve the sorting of strings, in the first case we sort rotations and in the second we sort suffixes. In fact, both concepts represent the same information; as it can be seen in Figure 6-4.

In consequence, a traditional way of generating the  $BWT(T)$  is via the suffix array  $SA(T)$ , as shown in the equation 6-1. In such cases, the SA construction represents the most consuming task in the generation of the BWT, taking around 90% of the runtime (as shown in Figure 6-2).

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] > 0 \\ \$ & \text{if } SA[i] = 0 \end{cases} \quad (6-1)$$

### 6.1.1.3. Suffix Array Construction (SAC)

SAs are lexicographically sorted data structures that contain all the suffixes of a given input. SA were introduced by Manber and Myers [180] as a space efficient alternative to suffix trees, in order to allow fast searches of patterns on very large texts (e.g. genomes). Since the SA is sorted, binary search can be used to look up which is commonly  $O(m \log(n))$ , where  $m$  is the seed length



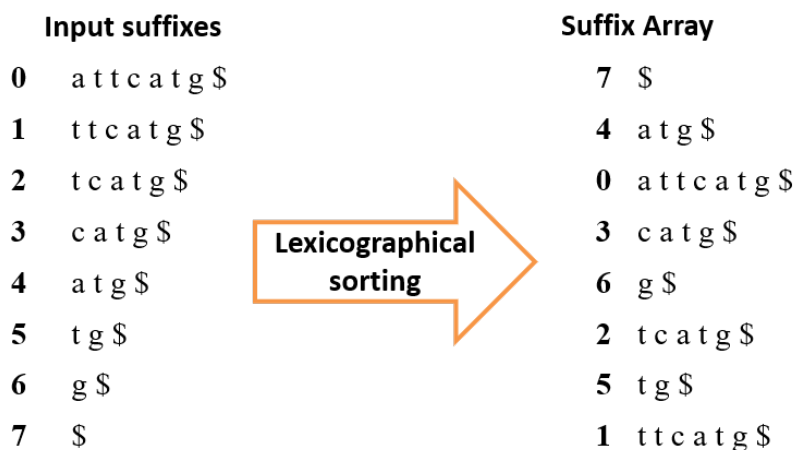
**Figure 6-3.** Example of calculating the BWM(T). In the left side all the permutations of T are shown. In the right side, the result of sorting lexicographically such permutations (Index *I*).

BWM:	Suffixes given by the SA
\$attcatg	7 \$
atg\$attc	4 atg\$
attcatg\$	0 attcatg\$
catg\$att	3 catg\$
g\$attcat	6 g\$
tcatg\$at	2 tcatg\$
tg\$attca	5 tg\$
ttcatg\$a	1 ttcatg\$

**Figure 6-4.** Comparison of the BWM(T) and the SA corresponding to that same input.

and  $n$  is the length of the whole input string [181]. SAs are widely used in full text indices, data compression algorithms, example-based machine translation, among many other applications.

There are three main classes of algorithms for SAC described in the literature: prefix doubling, recursive, and induced copying (sorting) algorithms. [178, 182]. Conceptually, all of them are focused on generating a set or sub-set of suffixes and then sorting them with different approaches. When applying SAC algorithms to genomic information, the huge amount of data involved and the dependences between them are the main challenges. In any case, the computational core and the most demanding task of the SAC is the suffix sorting (as shown in Figure 6-2). Figure 6-5 presents a simple example of the process of SAC.



**Figure 6-5.** Process of the suffix array construction for the input "attcatg". The suffixes are generated applying all the possible rotations of the input (left), and then those suffixes are lexicographically sorted using some specific algorithm (right).

Due to the reasons presented above, our acceleration efforts focused on the SAC, which has the most intensive computational work among the tasks inside the seed phase (Figure 6-2), the main bottleneck in UdeACompress (as showed in previous sections). Accelerating the SAC is particularly important to our research, not only because of the impact it has on the seeding phase, but also because the accelerated algorithms are also important in other parts of the workflow proposed in Chapter 4: (1) the Burrows Wheeler Transform which is carried out in the low level compression block (Figure 5.1.4), (2) the efficient characterization of the references in the database (performed off-line as explained in subsection 4.5) and the reads in the input (performed on-line), (3) the sorting of the reads prior to the encoding step inside UdeACompress, since a sorting algorithm is involved in the SA construction.

### 6.1.2. Assessing the Parallelization Feasibility in SAC

Sorting is the core, and the most time consuming part of SAC (Figure 6-2). The main problem of parallelizing many sorting algorithms is the high dependency among the operations. In the case of SAC there is an additional critical task: generating the suffixes from the original input (which we call Input Chars Generation). Only a single character per suffix is required on each iteration, so the input chars generation is executed sequentially at the expense of an extra computational cost; but providing a more memory efficient solution [182]. As we mentioned in section 2.3, we were interested in using a SIMD approach to accelerate the main bottlenecks in our algorithms, so in the next sections we will describe the process of developing a suffix array constructor.

It is well known that control structures in programs affect an efficient SIMD execution. Additionally, approaches of comparison-based sorting involve a high level of data dependencies, since each element has to be compared with many others. To avoid the effect of both factors, we selected a non-comparative method for the sorting of the large amount of suffixes involved. The state of the art on parallel sorting algorithms shows many works applying fine grain parallelism [131, 183, 184, 184–193]. We found interesting the work of Timothy Hayes in 2016 [79] since he compared the SIMD implementations of three different well known integer sorting algorithms: bitonic mergesort, recursive quicksort and LSD radixsort. This radixsort (named VSRsort) not only outperformed the rest of competitors in such comparison, but also scaled very well in different tests due to its hardware enhancements.

Nevertheless, there was an important limitation in VSRsort as its design was based on the use of two hardware instructions proposed by the author and not present on any commercial processor today: Vector Prior Instances (VPI) and Vector Last Unique (VLU). Hayes provided a simulated performance evaluation and description of his proposal [79], useful for any follow-up work.

As explained in section 5.1.1, UdeACompress uses the SDSL library [161] to build the FM-index required by the aligner. However, in this chapter, and based on the above analysis of the state of the art, we present our own design and implementation of a SIMD SAC that uses sorting approaches better better suited for SIMD programming.

Even though the interfaces of our SIMD SACs algorithms are compatible with the rest of the code in UdeACompress aligner, integrating them would also require to integrate the SIMD algorithms with the SDSL library. That involves a significant development and testing effort, which would not contribute significantly to our analysis. No negative impact on the results is expected because no significant overload would result from the aforementioned integration. Therefore, the algorithms presented in the coming section along with their experimental results correspond to isolated executions, in a stand alone setup.

### 6.1.3. SIMD SAC Design

In this section we explain the design and implementation of the SIMD SAC. First, we briefly discuss the main features of the selected architecture's ISA: The Intel AVX-512. Then we present the design details of our approach followed by the description of the algorithms implemented. This includes presenting two versions of our SIMD SAC algorithm: a naive implementation which incorporates basic optimizations and a version with more complex improvements. Finally, we discuss the issues related to the combination of SIMD and multi-threaded programming strategies to determine the feasibility of combining both approaches in the SAC algorithm.

Based upon the analysis presented in the Section 6.1.2, we set the following steps in order to achieved a SIMD-based parallelization of SAC:

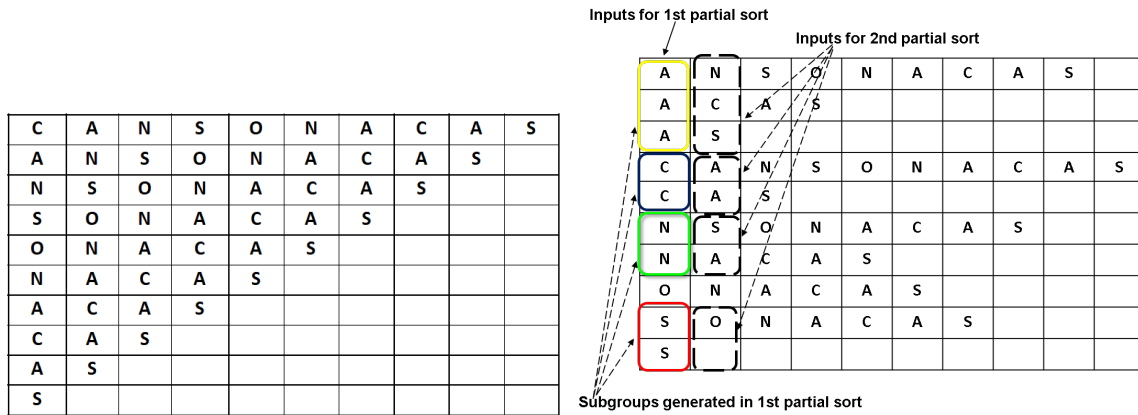
- Design software alternatives for the VLU and VPI operations proposed by Hayes, using the instructions available in Intel AVX-512.
- Develop a SIMD **Input Chars Generator** useful for further SAC construction.
- Implement a fine grain LSD Radixsort SIMD for positive integers.
- Integrate the previous codes to build a **SIMD SAC** algorithm.
- Evaluate the performance of the naive SIMD SAC in order to implement an optimized version.
- Compare the performance of both SAC methods.
- Determine the suitability of Intel AVX-512 for the efficient implementation of tasks associated with the construction of a SA.

#### 6.1.3.1. Design of the SIMD SAC algorithms

As stated before, our proposal for a SAC consisted in implementing two major task blocks: The first block generates the set of input characters to be used for the sorting in each iteration, one for each suffix. The second block performs a partial indirect integer sorting of the involved suffixes, according to the previously generated characters. This sorting is called indirect because we only sort the indexes instead of the suffixes, and it is called partial because it produces a partial sorting considering only the input characters corresponding to the current iteration; as such input changes iteratively, the final sorting is produced. Both of the blocks are combined under a "most significant character" approach, which processes each character from left to right. Three main steps describe our sorting process (presented in Figure 6-6):

- 1) Perform a 1st character regular suffix sort, using the first character of each suffix and performing the partial radixsort according to the ASCII value.
- 2) Define sub-groups of the previously unsorted suffixes which starts with the same letter. Update the input characters calculating the next most significant character for each suffix,
- 3) Separately, for each of the sub-groups with a cardinality greater than one found in (2), repeat the steps (1) and (2) until there are no sub-sets with a cardinality greater than 1.

One of the main challenges of our SIMD approach, was the sorting that needs to be performed continuously over the input characters. As mentioned in Section 6.1.2, after reviewing the state of the art we found that VSRsort for integer data suited well the sorting goals of our SAC, since the



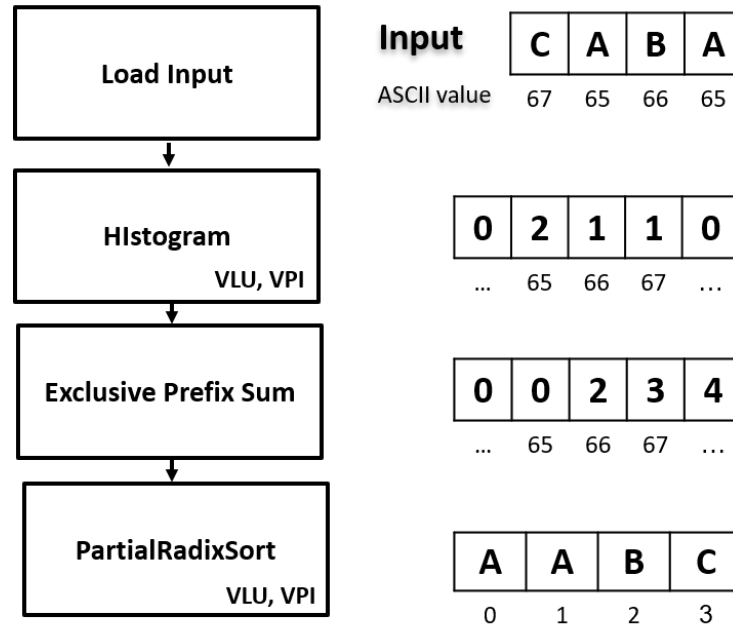
(a) Suffixes generated from the input string "CANSONACAS". (b) Identified sub-groups with the corresponding input characters for next iteration sorting

**Figure 6-6.** Example of the multi-level parallel Suffix Array Construction. First, all the suffixes generated from the input string are sorted considering the first character only. Then: (1) considering the characters sorted previously (currently in the 1st column), the suffixes are separated in subgroups starting with the same letter; (2) Each sub-group is sorted separately, considering the next right character of each suffix. The process is repeated moving to the next right character until there is no sub-group with a cardinality greater than one.

characters in the suffixes are associated to integer values according to the ASCII table.

In [80] Hayes presented a general description of VSRsort for sorting positive integers, with a complexity of  $O(k * n)$ , where  $k$  is an integer parameter that depends on the number of passes needed to iterate over all the digits of the longest integer in the input data and  $n$  is related to the amount of suffixes to sort. His proposal also included non-existing low level instructions to achieve a more efficient processing: VLU and VPI, both aiming at avoiding memory collisions in the algorithms. Based on the high level description of VSRsort we developed our own SIMD radix-sort for characters instead of integers and using native Intel AVX-512 instructions to emulate the VLU and VPI instructions. We named it PartialGroupSort because it sorts a specific group of suffixes partially, considering only the value of characters selected specifically for the current iteration. As seen in Figure 6-7, the overall idea is simple; it performs four steps sequentially. As discussed below, the SIMD PartialGroupSort is executed iteratively (as many times as the input size) over the different calculated groups to construct the final SA.

The first step (1) is selecting the corresponding input characters, one character per suffix. Let us assume the input array [C,A,B,A] as shown in Figure 6-7. In step (2) using the input ASCII values, a histogram is generated (characters counting). In the example shown: two A, one B and one C. In step(3) an exclusive prefix sum is calculated by adding all the previous elements for every position



**Figure 6-7.** Flowchart of the PartialGroupSort. In the right side, you can follow a simple example of each step, for the input array [C,A,B,A].

in the histogram (prefixing except for the current element). Finally, (4) using the VLI and VPU operations combined with the Exclusive Prefix sum, a sorting position is calculated for every input; which produces the partial sort.

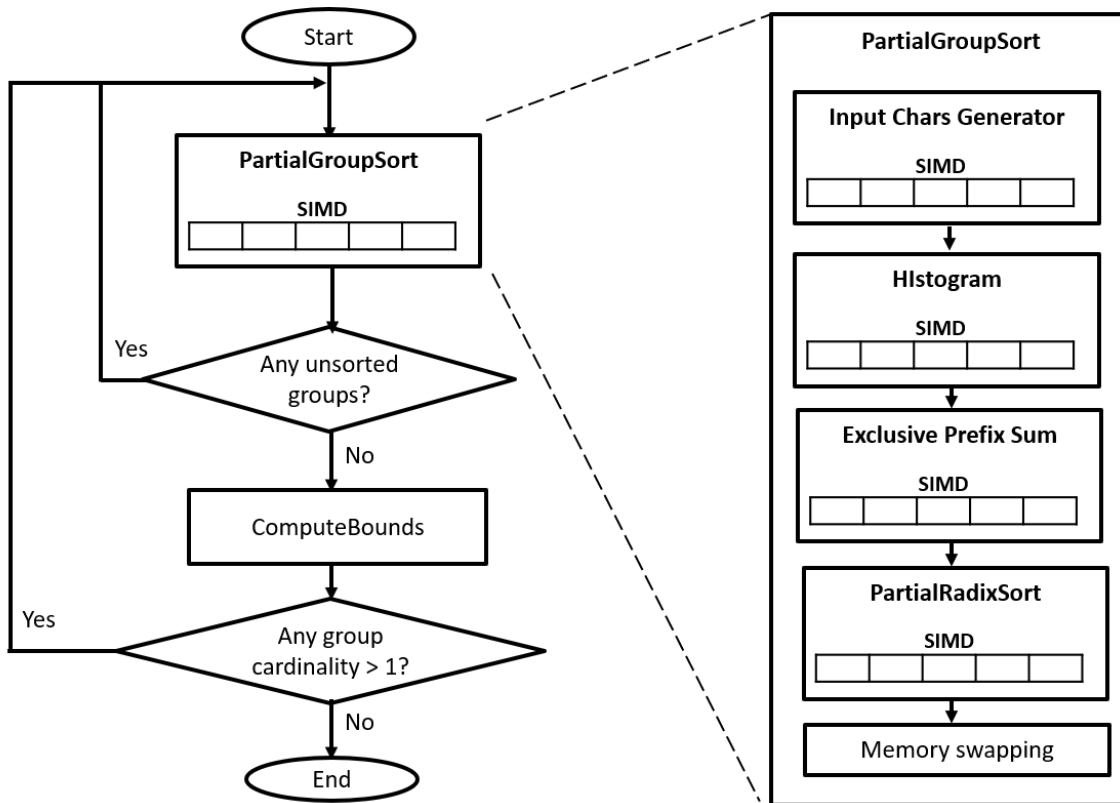
The new Intel's AVX-512 *Conflict Detection Instructions* were specially interesting to us, since they allow to detect elements with previous conflicts in a vector of indexes, generating a mask with a subset of elements that are guaranteed to be conflict free (non-repeated). A computation loop can be re-executed, and iterating over subsets of the remaining non-repeated elements until all the indexes have been operated will avoid access conflicts. Since we had the AVX-512 technology available, we used these instructions to design and implement the VLU and VPI operations.

### 6.1.3.2. Implementation of the SIMD SAC algorithm

In Figure 6-8 we show the complete process to efficiently construct the SA iteratively. The overall process consists of iteratively producing partial sorting per group of suffixes. In the first iteration all the suffixes form a single big group. Since at this point there is only one group, then we compute bounds among the groups produced by the recently performed partial sorting. After the bounds have been calculated we verify if there is any group with more than two suffixes, that is with a cardinality greater than one. The process is iteratively repeated: the inner cycle goes until all of the groups in the current iteration have been sorted, and the outer cycle when all the suffixes belong to a different group. In other words, the algorithm ends when all the groups have a cardinality equal



to one, which means that every suffix has been sorted.



**Figure 6-8.** Flowchart of the SIMD Suffix Array Construction Algorithm.

Groups bounds are calculated by the *ComputeBounds* function, which is responsible for calculating the different groups after the partial sort of every group through the iterations. Bounds are computed from the results of the last partial sort, separating in different groups those suffixes whose current input character is different. Once a suffix has been sorted, it stays in such position until the end of the process and it does not take part in any future relative sorting. SIMD improvements could not be incorporated to the implementation of *ComputeBounds*, since the SIMD approach did not suite the data dependencies and the flow of actions in it.

*PartialGroupSort* is responsible for the partial sort of every particular group, which happens sequentially through a series of steps analogous to VSRsort design. The input characters are calculated for the current selection of suffixes part of this group. Then a histogram is calculated over such input characters, and the result is used to compute the exclusive prefix sum. Then the partial radixsort is performed. Finally the memory positions are swapped so the indexes of the recently sorted array are updated. Then the execution continues in the main algorithm, building the groups to be sorted in the next iteration.

Next, we will present the details of implementation of the inner components of the partial group sorting. The specification of the low level data types and the instructions used in the following algorithms can be consulted in the Intel intrinsics guide.

### ***Input Chars Generator***

Input Chars Generator calculates the specific subset of characters that will be part of each current partial group sorting, one character per suffix. This is a consequence of not storing explicitly all the suffixes in an auxiliary table, in order to handle memory efficiently. This task requires considering three facts: (a) As seen in Figure 6-6, the set of suffixes to be sorted through the iterations process becomes smaller, (b) the suffixes change their positions continuously through different iterations as a consequence of each partial sorting and (c) the suffixes are not stored anywhere, instead they must be calculated and extracted from the original input. Our SIMD version of the *input chars generator* is presented in Algorithm 3<sup>1</sup>.

---

#### **Algorithm 3** Input Chars Generator

---

```

1: procedure INPUTCHARSGENERATOR(int Size, int *InputText, int currentIter, int *Indexes,
   int *OutInt)
2:   Displace ← CalculateDisplacement(Size, CurrentIteration)
3:   auxSuf ← *Indexes+Displace
4:   Displ ← _mm512_set1_epi32(Displace)
5:   for It←0;It<Size;It←It+16 do
6:     Suffix ← _mm512_load_epi32 ((*auxSuf+It))
7:     TextIndex ← _mm512_add_epi32(Suffix, Displ)
8:     Chars ← _mm512_i32gather_epi32 (TextIndex, InputText, 4)
9:     _mm512_store_epi32 (*Output+It,Chars)
10:  end for
11: end procedure

```

---

It starts calculating the displacement that must be applied to the original input to calculate the suffixes (based on the current iteration and the suffix position). An iterative process loads the input string, upon which the displacement is added, then the respective characters are gathered from the input string and finally stored in the output array.

### ***Histogram***

Many alternatives for implementing SIMD histograms have been proposed in the state of the art [194–200]. Hayes suggested in [80] that it was possible to calculate a histogram efficiently using the two non-existing instructions proposed in his research: VPI and VLU.

---

<sup>1</sup>The mnemonics used to express the instructions in the algorithms, as well as the respective description, can be found in <http://software.intel.com/sites/landingpage/IntrinsicsGuide/>

In our implementation, the input is loaded into a register, then the number of appearances of each element is calculated (using VLU and VPI) to generate a local histogram for such register. Finally, the affected positions are gathered from the global histogram, which is updated adding the current appearances for each value. A mask built based on the VLU is used to avoid collisions when scattering data.

A detailed description of VPI and VLU, along with our approach to implement such functions is presented next.

### ***Vector Last Unique and Vector Prior Instances***

One of the greatest challenges of Hayes proposal was the fact that it was based on two non-existing hardware instructions: VLU and VPI, both essential to build the integer radixsort. For that reason the priority was to implement both instructions using the instructions available in AVX-512 ISA.

The VLU instruction takes an integer vector in the input and produces a vector mask as a result. The idea is to mark the last instance of any particular value found. A bit in the output mask register is set only if the corresponding value in the input vector is not seen afterwards. On the other hand, VPI uses a single vector register as input, processes it serially and outputs another vector register as a result. Each element of the output asserts exactly how many instances of a value in the corresponding element of the input register have been seen before (excluding the current appearance). In both cases the elements in the input vector are processed from left to right [80].

After analyzing such instructions to find possible implementation choices in the AVX-512 ISA, we found that the recently released conflict detection instruction `_mm512_conflict_epi32` provided useful information to calculate VLU and VPI. Such instruction is barely documented, so useful information as the number of cycles to execute is unknown. The `_mm512_conflict_epi32` basically takes a register of sixteen 32 bits integers and returns a register of sixteen integers. Looking at a specific position in this output it is possible to know if the element in such position appears (are repeated) in a previous position of the input. Such information was useful to calculate both needed values: the last appearance of each element in the register and the previous appearances of the element in every input position.

Our implementation of VLU is presented in the Algorithm 4<sup>2</sup>. It takes the result of the conflict detection over the input and compresses it building a mask resulting from a bitwise disjunction among the integers in each position of the register. Then we calculate the bitwise exclusive OR between that mask and a constant 0xFFFF mask.

---

<sup>2</sup>The mnemonics used to express the instructions in the algorithms, as well as the respective description, can be found in <http://software.intel.com/sites/landingpage/IntrinsicsGuide/>

**Algorithm 4** Vector Last Unique

---

```

1: function VLU(_m512i Conf)
  ▷ Pre : Conf = _mm512_conflict_epi32(data)
2:   return( (_mmask16) _mm512_reduce_or_epi32(Conf) xor 0xFFFF )
3: end function

```

---

Considering the output of the `_mm512_conflict_epi32` instruction, our design for the VPI function was basically a pop-count over such register. By the time we implemented the VPI (see Algorithm 5<sup>3</sup>), the AVX-512 ISA did not have instructions for calculating the pop-count over a whole register. Our version for an efficient pop count was based in a lookup table to directly access the count value for each four bits integer. Input was separated into four registers, each filled only with four bits integers from original data, then the lookup table was used to get the number of ones in each case, which were added to get the final result. The look up table was built specifically to match our purpose and accessed using shuffle instructions. Even Intel has released recently a vector population count instruction (VPOPCOUNTDQ), it has been introduced only in the Knights Mill and Ice Lake architectures, which we had not available.

**Algorithm 5** Vector Prior Instances

---

```

1: function VPI(_m512i Conf)
  ▷ Pre : Conf = _mm512_conflict_epi32(data)
2:    $V_{0,3} \leftarrow \_mm512\_and\_si512(Conf, 0xF)$ 
3:    $V_{4,7} \leftarrow \_mm512\_and\_si512(\_mm512\_srli\_epi32(Conf, 4), 0xF)$ 
4:    $V_{8,11} \leftarrow \_mm512\_and\_si512(\_mm512\_srli\_epi32(Conf, 8), 0xF)$ 
5:    $V_{12,15} \leftarrow \_mm512\_and\_si512(\_mm512\_srli\_epi32(Conf, 12), 0xF)$ 

6:    $cnt_1 \leftarrow \_mm512\_shuffle\_epi8(\text{lookup}, V_{0,3})$ 
7:    $cnt_2 \leftarrow \_mm512\_shuffle\_epi8(\text{lookup}, V_{4,7})$ 
8:    $cnt_3 \leftarrow \_mm512\_shuffle\_epi8(\text{lookup}, V_{8,11})$ 
9:    $cnt_4 \leftarrow \_mm512\_shuffle\_epi8(\text{lookup}, V_{12,15})$ 

10:  return(  $\_mm512\_add\_epi32(\_mm512\_add\_epi32(cnt_1, cnt_2), \_mm512\_add\_epi32(cnt_3, cnt_4))$  )
11: end function

```

---

As explained before, the new `_mm512_conflict_epi32` was essential for implementing VLU and VPU. Nevertheless, it was designed to receive as input only registers of sixteen 32 bits integers;

<sup>3</sup>The mnemonics used to express the instructions in the algorithms, as well as the respective description, can be found in <http://software.intel.com/sites/landingpage/IntrinsicsGuide/>

which had an important impact in the overall performance of the radixsort. In the case of SAC, the optimal performance can be achieved if sixty four 8 bits integers can be processed at the time. This was the greatest bottleneck of our SIMD program, limiting the performance of the rest of our algorithm by up to  $4\times$ . Additionally, this required that, at the beginning of our algorithm each character in the original input had to be expanded from 8 to 32 bits, with the respective overhead.

### *Exclusive Prefix Sum*

Our method for calculating the exclusive prefix-sum iterates over the whole histogram: loading every 16 elements into a register, calculating the local exclusive prefix sum over such register and updating the exclusive prefix sum array. The last element in the local exclusive prefix sum register of the current iteration is carried to update the values of the next one. The local prefix sum is calculated combining cumulative additions and shifting operations a fixed distance: 1, 2, 4, 8; in the form of a tree. The shifting separates the integers in the input until we have four arrays with four bit elements in each. The number of bits of each element is directly obtained from a look-up table and then totalized to obtain the final value.

### *Partial Radixsort*

Our partial radixsort (Algorithm 6<sup>4</sup>) performs an indirect sorting, which means the suffix indexes are sorted instead of the suffixes themselves. The partial radixsort iterates over registers with 16 input characters performing the following steps: first VPI and VLU are re-calculated, then the VPI is incremented in one (to update the offset of the current element). The exclusive prefix sum is gathered from the input, and this register is added to the VPI value in order to obtain the exact offset to be place each element to produce the partial final sorting. Then, the corresponding suffixes are loaded to scatter them according to the previously calculated offsets. Finally, the exclusive prefix sum array is updated for future iterations.

#### **6.1.3.3. Optimizing SIMD Suffix Array Construction**

In the first version of the SIMD SAC code, named SIMD\_Naive, we applied some strategies oriented to optimize performance:

- Minimizing scalar operations for a maximum harnessing of the SIMD architecture,
- Inlining all functions to reduce execution overhead,
- Padding input data to avoid scalar processing, when the amount of data was below registers length,

---

<sup>4</sup>The mnemonics used to express the instructions in the algorithms, as well as the respective description, can be found in <http://software.intel.com/sites/landingpage/IntrinsicsGuide/>

**Algorithm 6** Partial Radixsort

---

```

1: procedure PARTIALRADIXSORT( (int *InputChars, int Size, int *ExPsum, int *Indexes, int
   *SortedIndex) )
2:   for x←0;x<Size;x←x+16 do
3:     data ← _mm512_load_epi32 (*InputChars+x)
4:     outConf ← _mm512_conflict_epi32(data)
5:     VPI ← VPI(outConf)
6:     mymask← VLU(outConf)
7:     auxPsum← _mm512_i32gather_epi32 (data, ExPsum, 4)
8:     offsets← _mm512_add_epi32 (auxPsum, VPI)
9:     updatePsum← _mm512_add_epi32 (offsets, Reg_One_AVX512))
10:    suff_Indexes ← _mm512_load_epi32( *Indexes+x)
11:    _mm512_i32scatter_epi32 ( SortedIndex, offsets,suff_Indexes, 4)
12:    _mm512_mask_i32scatter_epi32 (ExPsum, mymask, data, updatePsum, 4)
13:   end for
14: end procedure

```

---

- Carefully implementing VLU and VPI instructions, using the minimal amount of native instructions.

However, we also report the results of applying additional optimizations to overcome important drawbacks identified in the first version of the SIMD SAC. The ones with greater impact in the performance were:

- The impact of the *conflict detection instruction*. Not only because of the complexity of such instruction per-se (the number of cycles required by the instruction was not documented by then), but also because of its interface. The instruction was designed to process strictly 32 or 64 bits integers, while we would rather process sixty four 8 bits integer to harness all the hardware capabilities. In consequence, performance could be around  $4\times$  lower than expected. Additionally, this issue required to expand the input data before processing it, in order to convert it from 8 bits to 32 bits, causing an additional overhead. Our strategy to tackle this problem was packing two characters into a single 32 bit integer. Even though this clearly would help to improve performance since the SAC could save up to half of the iterations, it would also introduce more overhead since the expand task needed to be more complex, involving extra bits displacement and adding operations. This strategy could also have a negative impact since the histogram and the exclusive prefix sum will have to handle bigger structures, leading to a greater number of cache misses when continually performing the mostly sparse data access involved to calculate the histogram, the exclusive prefix sum and the partial sorting.

- The problem of indirect sorting. SIMD approach performs much better when processing contiguous data. But performance may decrease significantly when sorting dispersed elements due to the amount of cache misses. It is difficult to efficiently implement an indirect sorting approach using SIMD instructions because the indexes to be sorted are associated to data that is not stored contiguously in memory; in consequence costly gather/scatter operations to load the keys into the vector registers had to be used. Such overhead could offset the benefits we gain from the usage of SIMD instructions. To reduce the impact of this issue, we implemented two different versions of each function: apart from the previously discussed regular versions (used in the iterative process), we created specific versions to perform contiguous memory access (instead of dispersed) when possible, specifically in the first iteration. In the first iteration all the functions manipulate the greatest amount of data since the input corresponds to the whole input, while in the rest of iterations each function manipulates the data of each sub-group (clearly with less elements). Also, with this change we avoided invoking the input chars generator the first time (they correspond directly through initial consecutive indexes), and a high cost *memcpy* operation which was replaced by a pointer swapping over the sorted indexes data.
- In our SAC we calculated the VLU and VPI instruction twice over the same input: once during the histogram and a second time when performing the partial radixsort. This was optimized by storing both values after they are calculated in the histogram, and loading them at the beginning of the partial radixsort; at the expense of higher memory costs.
- During comprehensive testing we realized that groups size decreased quickly, and an important percentage of the time was spent in the scalar sorting of small groups. This issue is commonly tackled through padding, but in our case the application of padding was limited because the groups data was processed independently but stored contiguously, leaving no space for padding. In consequence, groups with only two suffixes are sorted directly using scalar comparisons only.

From now on, we refer to the algorithm implementing the aforementioned optimizations as `SIMD_OPT`.

## 6.2. Performance Evaluation

In this section we present the performance tests of the three SAC versions developed:

- **Scalar:** A purely scalar SAC algorithm to be used as a baseline. Scalar is also used as a reference to calculate the speedup of the respective parallel implementations. The sorting

kernel in scalar implements an efficient radixsort as presented by [201]. When less than 120 elements are to be sorted, Scalar uses an optimized insertion sort algorithm instead, which is more efficient (according to our former experiments).

- **SIMD\_Naive**: The initial version of the SIMD SAC, incorporating basic optimizations as explained in the previous section.
- **SIMD\_OPT**: The final optimized version of the SIMD SAC, incorporating elaborated optimizations as explained in the previous section.

The three SAC versions have a complexity  $O(n^2)$ . We did not modify in the strategies of such methods oriented to change their complexity, since our main interest in this phase was to explore the effects of SIMD parallelism in the final runtime. In consequence, the computational complexity across the three versions does not change.

Tests were performed using DNA data extracted from the genome hg19<sup>5</sup>. Such input was preprocessed to fit into the alphabet (A, C, G, T, N). We had the hypothesis that our SAC algorithm was not affected by the characters distribution, because it does not consider any relative pre-existing order in the input. However, to evaluate the effect of the input distribution in our method we used different chromosomes in hg19 to create different input files. As the performance variation was statistically non-significant, it was clear that none of the developed method was affected by the bases distributions in the different inputs,. As this behaviour was consistently exhibited through all the experiments, the rest of our analysis is made considering the first input only.

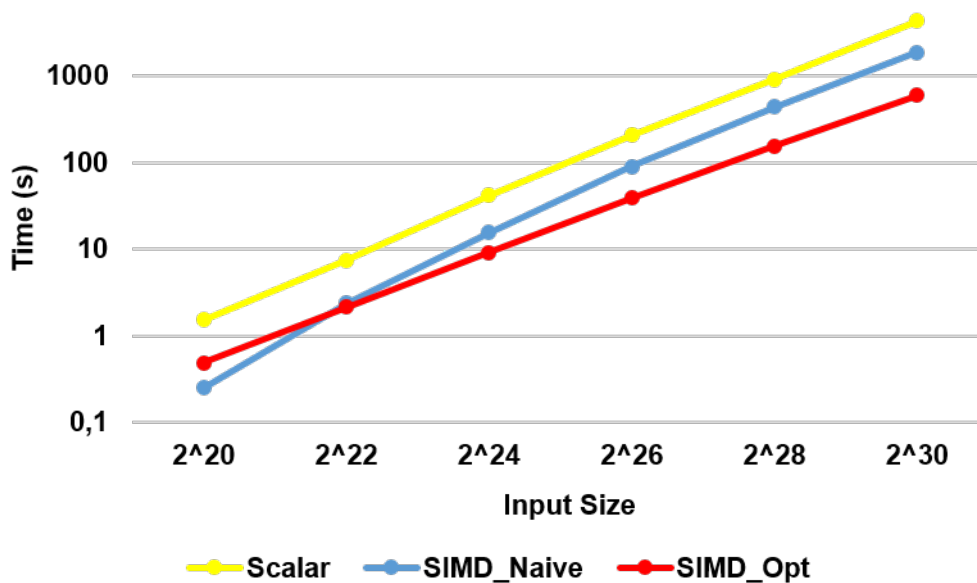
Tests were performed on a server with two Intel(R) Xeon(R) CPU 6152, 2.10GHz, 30 MB cache, 192 GB of RAM DDR4 in a shared memory architecture, with a 4 TB SATA 3.5" Hard Drive at 5400 RPM, and using Ubuntu 16.04 OS (64 bits). We report the minimum time of three replicas executed, measured with `chrono::high_resolution_clock`. All codes were written in C and compiled with Intel C++ compiler version 18.0.2. using optimization flag `-O3`.

### 6.2.1. SIMD SAC Performance

Figure 6-9 shows the runtime comparison of both SIMD versions against our baseline. Both of our SIMD versions were consistently faster than the scalar baseline. For small inputs (4 million characters or less), the SIMD\_Naive version was up to  $3\times$  faster than SIMD\_OPT. We hypothesize that since the data structures size grew along with the input size, this generated in a higher probability of cache misses due to memory access that are made more frequently and more scattered. This cumulative effect resulted in SIMD\_Naive being less efficient as input size grew. The general greater performance of SIMD\_OPT was also because of the data packing optimization applied, which reduced by half the number of iterations in the inner cycle.

<sup>5</sup><http://hgdownload.cse.ucsc.edu/goldenpath/hg19/bigZips/chromFa.tar.gz>



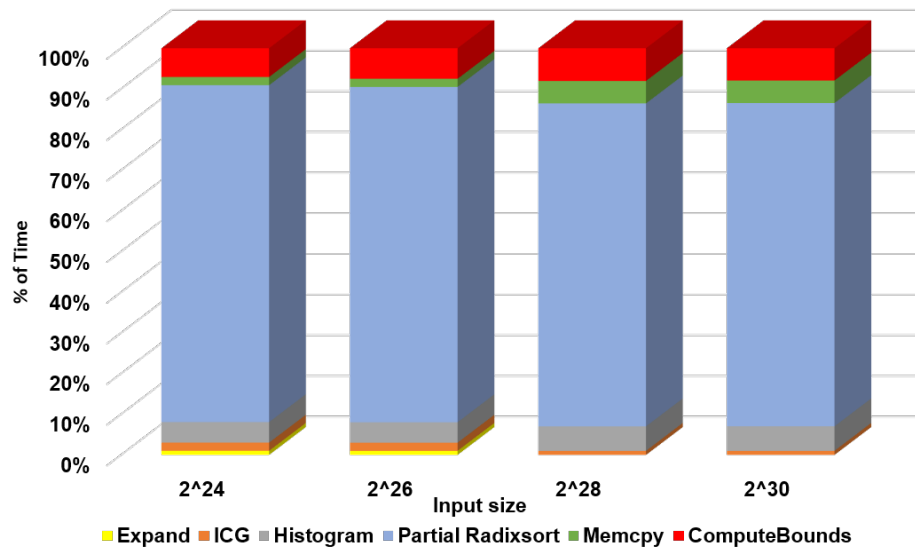


**Figure 6-9.** SAC execution time with different input sizes from hg19. Vertical axis is  $\log_{10}$  and the horizontal is  $\log_2$

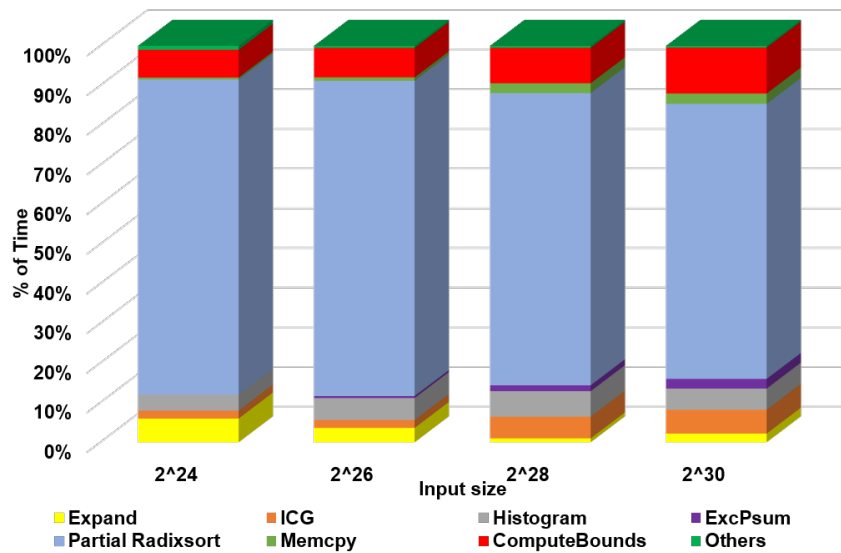
For very large input sizes (more than  $2^{24}$  characters), SIMD.OPT was between  $2\times$  and  $3\times$  faster than SIMD.Naive. For small inputs ( $2^{22}$  or less characters), the SIMD.Naive performed better than the SIMD.OPT version. In the test with the largest number of characters, corresponding to bases  $2^{30}$ , the difference between the baseline and the SIMD.OPT was approximately one order of magnitude.

It was very important to measure the impact of every function inside our algorithms in the global performance of our SIMD SAC. This way we could observe the effect of the implemented optimizations and identify other optimization objectives for the future. The profiling results for the larger input sizes are shown in Figure 6-10.

In SIMD.Naive the tasks that dominate the performance are evident: mainly partial radixsort, and with significant less importance the histogram computation, *ComputeBounds* and the *memcpy*. Meanwhile, in the SIMD.OPT version we observe the effect of additional tasks that now become important as the time required by the most expensive functions is reduced. The dark green block in the SIMD.OPT profile, which did not appear in the naive implementation, represents mostly the functions that only perform contiguous memory access (non gather/scatter) which are executed in the first iteration, along with the cost of the scalar direct sorting of groups with only two suffixes. According to additional experiments, functions that only performed contiguous memory access were at least  $2\times$  faster than the counterpart gather/scatter based functions.



(a) SIMD\_Naive SAC algorithm.



(b) SIMD\_OPT SAC algorithm

**Figure 6-10.** Profiling of both versions of the SIMD SAC algorithm.

The second optimization included was storing the VPI and VLU (instead of just recalculating it), whose effect is observed when comparing the histogram block of both figures (in gray). The gray blocks in the profile of SIMD\_Naive tend to be smaller than in the SIMD\_OPT, where more time is consumed due to the additional store instructions that are performed. This also contributed to reducing the runtime of the Partial Radix. This difference was more evident when the input sizes increased. This also reduced the runtime of the partial radixsort since it now does not calculate the VLU and VPI.

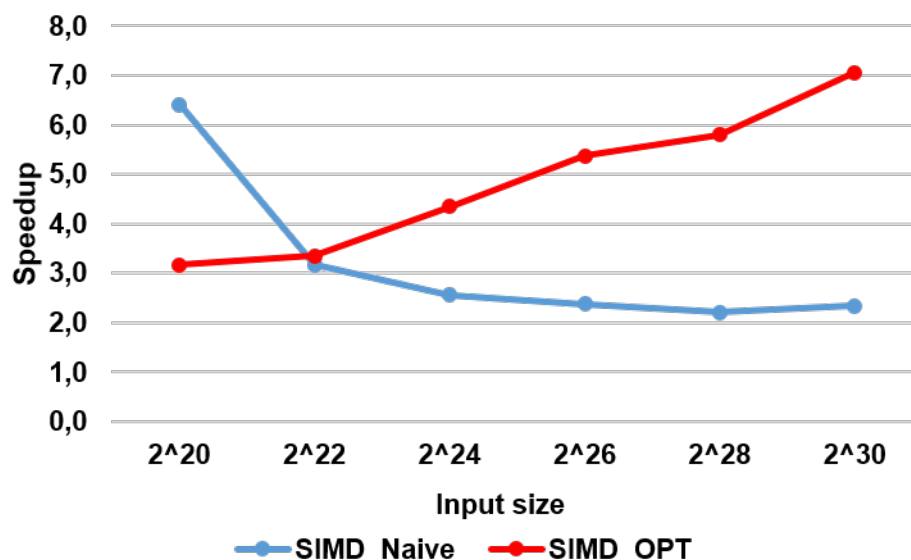
The effect of the packing strategy is observed in partial radix blocks, which consumes less percentage of the total runtime in the optimized version. Packing also reduced the number of iterations of the inner cycle, which also resulted in less time spent in the *memcpy* instructions. This positive effect might be offset in the case of the histogram block because of the overhead introduced to store the VLU and VPI arrays. Also, both the histogram and the exclusive prefix sum are negatively impacted since now they must handle much larger data structures (more buckets are needed), increasing the probability of cache misses in both tasks. Additionally, the cost of the expanding function became significant in the SIMD\_OPT version, because in this version the expanding process involves more complex operations; and because of the general impact of some other functions had been reduced significantly due to the packing itself. Due to this runtime decrease, the non-SIMD portion of the code (mostly the scalar direct sorting and *ComputeBounds*), exhibited a greater impact in the overall performance. Finally, the observed reduction in the execution time of the partial radixsort is also a consequence of the scalar direct sorting of groups of size 2, which had a higher impact for small inputs. These findings also explain the variation in runtime time shown in Figure 6-9.

Finally we calculated the performance gain of both of the SAC algorithms, which is presented in Figure 6-11. Considering the Intel AVX-512 architecture used, the ideal parallelism should correspond to a speedup of  $16\times$  for 32-bit element arrays. Figure 6-11 shows that, on one hand, SIMD\_Naive achieved a speedup of up to  $6\times$  for small inputs, which decreased as the input size increased. On the other hand, SIMD\_OPT exhibited a speedup that grows with the input, up to  $7\times$  faster than the reference, for the experiments presented. Such speedup can be considered as significant taking into consideration the overhead and implementation issues related to SIMD programming.

Although one of the optimizations applied was to pack two characters into one array element, we decided not to try for a denser packing strategy. We think that by doing so, it would also result in a negative impact in the final performance, since with bigger integers the histogram and the exclusive prefix sum would need much more buckets. This is translated into much higher probability of cache misses, with an important cumulative effect because of all the scattered accesses. The most convenient solution for this issue would be having a more versatile *conflict detection* instruction that would allow processing more elements (of less size) per AVX-512 register.

### 6.2.2. Considerations for Instruction-set Extensions

Based on the presented experiments, we identified some potential instructions that can be considered by processor designers in future ISA extensions. Such instructions would help improving performance, not only for problems such as the SAC presented here, but also in tasks related to



**Figure 6-11.** Speedup of SIMD\_Naive and SIMD\_OPT against Scalar.

sorting, multimedia processing and signal processing, among others types of applications. Beyond the VLU and VPI instructions proposed by Hayes we also propose the following:

- VPopCnt: Vector Pop Count. As we have mentioned, Intel released instructions for vector population count (VPOPCNTDQ extension), they have been introduced only in the Knights Mill (2017) and Ice Lake (2019) architectures, but they were not incorporated to the other processors like the Cannon Lake released in 2018.
- VPrefixSUM: Vector Prefix Sum, an instruction to compute the prefix sum of the input register.
- VHistogram: Vector Histogram, an instruction to compute the histogram of the input register.
- ConflictDetection\_32x16: specific version of the conflict detection instruction, but designed to process 32 x 16 bits integers.
- ConflictDetection\_64x8: specific version of the conflict detection instruction, but designed to process 64 x 8 bits integers (signed and unsigned).
- Shuffle\_RegToReg\_epi32 (`_mm512i` in, `_mm512i` out, `_mm128i` pos): This instruction represents a friendly version of the intrinsic to shuffle 32 bits integer on a AVX-512 register (*in*) to specific positions (*pos*) in another AVX-512 register (*out*). Even though this type of task is currently possibly using shuffle or permute intrinsics, their usage can be very tricky requiring the programmer to handle many low level details.

Nevertheless, the hardware cost of such instructions needs to be determined in order to decide the viability of including them into commercial processors.

### 6.2.3. Introducing Thread Level Parallelism into the SIMD Approach

After achieving a successful acceleration our implementation of SIMD SAC, we were interested in testing the effects of combining the SIMD strategy with TLP, in order to determine the suitability of combining both approaches in our proposal for SAC. Two additional versions of SAC were developed:

- An OpenMP parallelization of the scalar version (non-SIMD).
- AN OpenMP parallelization of the SIMD\_OPT version.

OpenMP directives to produce parallel threads were inserted into the partial groups sorting, this is, inside of the input chars generator and inside of each of the process corresponding to the group sorting (Histogram calculation, and partial radixsort). Other minor SIMD tasks (expand and suffix initialization) were also parallelized using OpenMP threads. The suffixes inside each group were sorted using threads, but the different groups were sorted sequentially in this approach.

From the firsts experiments of both versions, we observed that the scalability was poor. Profiling both performances we found that most of the running time was spent in system processes. Threads were executed in a very small percentage of total runtime. After researching, we identified a problem of non-uniform memory access. Considering that OpenMP is not NUMA aware (during runtime the threads are just dispatched to any available CPU), we tried executing the program modifying the openMP affinity through the KMP\_AFFINITY flag, in order to see the effect in performance. We observed a little gain in performance oscillating between 5% and 7% in overall performance when setting the affinity parameter to "compact", which causes the threads to be placed as close as possible. We also configured the OpenMP environment variable OMP\_PROC\_BIND to control binding of OpenMP threads, so that the OpenMP execution environment would not switch threads among processors. Performance did not change significantly.

A hypothesis we propose is that the inner design of our SIMD SAC is not cache friendly. One of the main reasons is that we perform an indirect sorting instead of a direct sorting. Because of this, very large data structures are accessed and modified through gather and scatter operations. This is the only way of avoiding the comparisons required to perform data sorting. Such data structures (histogram, exclusive prefix sum, indexes and input characters) are constantly updated during the most time consuming functions in our algorithms. This NUMA became significantly evident in the multi-threaded implementation, showing a bigger impact in runtime. As the number of threads was

increased and the memory accesses became even more scattered, the benefits of multi-threading were not enough. In such scenario, many threads could update contiguous memory positions belonging to the same block, invalidating cache content of the other threads. Then, keeping the cache coherence becomes more complex, generating much more traffic, and resulting in more cache misses. To tackle this issue we would have to implement the multi-threaded parallelism at groups level, which means each group is sorted with a single thread and using SIMD\_OPT only; since this would reduce significantly the memory sharing between different threads.

### 6.3. Summary

In this chapter we presented the strategies implemented to accelerate the execution of UdeACompress. Considering the results of a profiling discussed in chapter 5, we implemented parallel algorithms that accelerate the most consuming task in UdeACompress performance: The SA construction. Such task, essential for the seeding phase of the alignment, was accelerated through DLP. The optimization was done in isolation and not incorporated inside the original code of UdeACompress.

The AVX-512 ISA was extensively studied to select the proper instructions to implement a SIMD SA Constructor. VPI and VLU operations proposed in a previous work were implemented with the latest instructions available in Intel AVX-512.

Two versions of SIMD SAC were developed: a naive implementation with some basic optimizations, and a more elaborated version with further low level optimizations. The optimized version achieved up to  $7\times$  of speedup against the scalar version when processing larger inputs.

The non-comparative sorting approach suited very well the DLP parallelism avoiding multiple comparisons and excessive control flow. However, this approach presented some important issues. It does not harness any relative pre-existing order in input data; leading to unnecessary data movement. Also, it is not very cache friendly because of the constant scattered memory accesses to large data structures. Both negative effects were exacerbated because of the indirect sorting strategy applied.

Although, the use of the conflict detection instruction to emulate the VLU and VPI instructions worked out well, the available element sizes limited the achievable performance.

A set of useful ISA instructions was proposed for the acceleration of SAC.

Finally, two multi-threaded versions of SAC were developed with preliminary results showing

negative performance results. The gather-scatter memory access patterns to large data structures created a very inefficient use of cache when involving multiple cores and threads. This issue is left as open for future research.

The work presented in this chapter is to be partially published in:

**Guerra, A.,** Quintero, J., and Isaza, S. (2018). Efficient SIMD Parallelism for Suffix Array Construction. Manuscript to be submitted to "Parallel Computing", Elsevier.

## 7. Conclusions and Further Research

Advances in sequencing technologies in the last years have caused a dramatical reduction of the costs related to genome sequencing, leading to a continued exponential growth of the genomic data available. Even though specialized compression strategies have arisen as useful alternatives to tackle this issue, they are not mature yet and exhibit great computational challenges as well. As an answer, in this dissertation we presented a contribution to the field of compression algorithms for Next Generation Sequencing data in FASTQ format.

First we introduced the most important concepts related to bioinformatics and high performance computing architectures as a necessary background for understanding the rest of this document. Then, we presented a comprehensive review and performance evaluation of the state of the art in DNA compression algorithms, highlighting the most important weaknesses found and discussing the opportunities for improvement. In the following chapters we presented the design of a workflow for FASTQ referential compression, the implementation and evaluation of the workflow's core (a multi-technique compression scheme which implements a referential compressor for reads), and the efforts made to accelerate the execution of the proposed algorithms through parallel programming.

In this chapter we summarize the work done and present the respective conclusions. Last, we propose possible directions for future work.

### **State of the art**

A literature review of FASTQ compression algorithms led us to notice a significant body of works on non-referential strategies. In Chapter 3 we made a comprehensive review and an experimental quantitative comparison the related works, allowing us to identify the big challenges and opportunities in the field. The detailed evaluation included relevant metrics as compression ratio, compression and decompression throughput, parallel scalability, and peak and average memory consumption. We found many interesting facts:

- The higher compression ratios achieved through specialized algorithms, taking advantage of the specific properties of genomic data; at expense of a slower and more memory consuming performance.
- Despite exhibiting lower compression capabilities, general purpose compressors were com-



monly used for genomic data; do to their ease of use and compatibility with different file formats.

- The top specialized compressors combined simultaneously different strategies, and treated separately each stream of the FASTQ file.
- The use of HPC in specialized compressors was incipient, absent in some cases or showing a poor scalability in others.
- By the moment of such evaluation (2014), we did not find reports of referential compression for FASTQ files in the literature.

At the end of Chapter 3, we also presented a review of the state of the art of referential compressors for other different types of genomic data. We found that the performance of the top non-referential compressors was far below from what a referential compressor could theoretically achieve. However, the application of the referential approach had been limited by the need of an appropriate reference and by the high computational cost of the involved tasks.

The comprehensive study presented in this chapter allowed us to characterize the problem, identifying the need of algorithms for efficient referential compression of FASTQ files. In order to increase the usability of such algorithm two issues had to be addressed: proposing solutions for the selection of an appropriate reference and the harnessing of HPC resources to accelerate the execution.

### **Referential compression workflow**

We realized that referential compressors had been successfully used for the compression of genomes. However, some challenges limited the application of that approach for the compression of FASTQ reads. To overcome the problem of the reference selection, in Chapter 4 we presented the design of a compression workflow for the automated selection of an appropriate reference. Each block was specified in detail, as well as the data-flow among the different tasks involved.

The workflow was intended to find the minimum mathematical distance between inputs and each reference, through two basic stages previous to the compression: input features detection and then classifying such input into a set of stored references. This would benefit not only the usability of the compressor, but also the compression itself. In a third stage, the compression is performed in the multi-technique compression scheme, where the different streams inside the FASTQ file are processed through very specialized strategies. The core of such scheme was the referential reads compression, based on a read-to-reference alignment combined with reads reordering and an elaborate binary encoding mechanism.

Even though in this thesis we did not present the full implementation of the proposed workflow, we did contribute theoretical evidence of the feasibility of implementing them successfully. For feature detection the features text analysis based on the FM-Index was not only accurate and efficient, but also associated with other tasks executed in the others workflow's blocks. On the other side, relevant ideas of machine learning showed to fit the requirements of sequences classification. Additionally, we introduced the guidelines to be considered when creating the references database. Finally, the basic issues related to the compression of identifiers and quality scores were discussed, tasks which were carried out by third-party software.

As a result of this design, the most complex and relevant tasks of the workflow became our main objective. We concentrated the efforts on the development of the blocks inside the multi-technique compression scheme; specially on the referential compressor for FASTQ read sequences, as the main objective of this thesis.

### **FASTQ referential compression**

In Chapter 5 we presented the implementation of UdeACompress: a FASTQ referential compressor and the core of the multi-technique compression scheme presented in the previous chapter. UdeACompress consists of a set of blocks responsible for: specialized alignment between the reads and the reference, sorting the reads according to their mapping position, the encoding of the alignment data through a map and a set of instructions, and the low level compression of such encoded information. Among the strategies applied to maximize the encoding efficiency, new matchings and mutations were introduced, as well as an assignment of the instruction codes based on the probability of occurrence of each transformation in real datasets.

During the development of this thesis, the first referential compressors for FASTQ began to appear, demonstrating the importance of the research topic. We studied an run experiments to compare those works with our own, along with other top specialized non-referential compressors. Several performance metrics were taken from tests on real datasets, where our proposal achieved results at the same level of the state of the art compressors, in terms of compression capabilities and decompression speed (the most important for usability reasons). Although compression times and memory consumption of UdeACompress were high, they were competitive in comparison to the other applications and acceptable considering the architectures commonly available in bioinformatics facilities. Finally, a simple model resulting from profiling UdeACompress allowed us to estimate a noticeable reduction in compression runtime in a parallel execution scenario.

Some weaknesses were identified in the compressor: the aligner was still unable to map a significant percentage of reads, decreasing the efficiency of the compression. This is also related to the compression capabilities of the unmapped reads, which were poor compared to the performance of the referential compressor. Also, the compression of quality scores exhibited an important impact

in the compression ratio of the whole FASTQ file content, requiring finer approaches if higher compression ratios are to be achieved.

As UdeACompress was envisioned thinking in an efficient compression of long reads, simulated data was generated to measure the compression capabilities of long read sequences. In tests with different coverages, read lengths and number of mutations per read; UdeACompress compression was always superior to the any algorithm in the state of the art.

### **Improving execution time**

After evaluating the performance results, we set to accelerate the main bottleneck found in the profiling of UdeACompress, the aligner, taking advantage of today's high end processors. In Chapter 6 we presented a detailed analysis of the tasks inside the seed block of the aligner, which led us to identify the construction of a Suffix Array as the main bottleneck. Furthermore, accelerating the Suffix Array Construction was considered important not only because of the alignment; but also because of its use in other parts of the proposed workflow: the low level BTW compression, feature detections (on genomes and reads) and reads sorting.

The SAC algorithm was designed based on an indirect sorting approach, and implemented using the Intel AVX-512 instruction set extensions. We achieved a speedup of up to  $7\times$  in the SAC, which could lead to an important reduction of the overall execution time in UdeACompress. Since the SIMD implementation is tied to the instructions provided by the ISA, finding the appropriate instructions to implement each task was one of the greatest challenges, in order to achieve acceptable levels of speedup. Although considerably high speedup were achieved, some of the limitations we found were: the great amount of scattered memory access in our approach and the interface limitations in some of the low level instructions required.

Profiling different versions of the SIMD SAC algorithm, we measured the impact of different manual optimizations. Such optimizations tended to be tricky, demanding important programming abilities and developing time in order to handle low-level details. The most relevant were: avoiding control flow, avoiding memory collisions, maximum reduction of scalar instructions, data packing, and preferring contiguous memory access.

Intel's recently released conflict detection instruction was a very convenient choice to calculate the repeated elements in a register, which we used to avoid memory collisions. Nonetheless, its programming interface restrictions for value sizes imposed limitations on the achievable performance. Anyway, the conflict detection instruction was found very useful and we consider it will contribute to accelerate a large range of problems of high computational complexity.

The implementation of the SIMD SAC algorithm also required the emulation of essential opera-

tions that were only simulated in the previous work we were based on, and that were not present in today's Intel AVX-512. After our experience with the manual low level implementation of the algorithm, we also proposed further instruction set extensions.

Finally, preliminary tests of the combination of coarse (thread level) and fine grain (SIMD) parallelism were performed. Such experiments did not show satisfactory results, exhibiting poor scalability. The impact of the main drawbacks in our approach, which was barely observed in the SIMD implementation, was exacerbated by the non-uniform memory access during the multi-threading processing, leaving some work for future research.

We believe that the detailed review and comparative analysis of the state of the art, the design of the workflow for referential compression of FASTQ files, the compressor that has been discussed and implemented, and the achievements and learning from the SIMD developments; represent an important contribution to the clarification and solution of the most important challenges of efficiently storing NGS raw data, which will allow the development of better solutions in the future.

## **Future Work**

In spite of the development efforts, encouraging results and insights provided by this thesis, there is still ample work to be done in order to achieve maturity in compression of FASTQ files. This very last section discusses the possible research directions to complement this thesis.

To improve the compression capabilities of UdeACompress three main issues should be addressed. It is necessary to refine the design of the aligner in order to increase the amount of mapped reads. An additional set of matchings could be statistically evaluated and incorporated in the search process, as well as heuristics to direct the search closer to the optimal alignment. Another interesting development would be to apply statistical techniques for the construction of pseudo-genomes. They could be built based on common sub-strings extracted from each species of interest for the reference database. Doing so could help improving significantly the similarity between reads and reference.

The second issue is to boost the capabilities of the low level compressor (used for encoded data and unmapped reads), implementing specialized strategies for each case. In both cases, delta encoding and Markovian predictive models could help enhance the current algorithms. Also, approaches based in k-mers features (e.g. overlapping reads, graphs) could also have a positive impact for the compression of unmapped reads. Finally, implementing the two first stages of the workflow (feature detection and classification) may lead to the selection of references of higher similarity, even beyond phylogenetic relationships. However, solving such issues will result in more intensive

computation.

The compression speed of UdeACompress was acceptable in comparison to the state-of-the-art, but with an important space for optimization. Two important enhancements remain to be applied: the SIMD SAC algorithm (presented in Chapter 6) should be integrated in the aligner. Also, the encoder block should be parallelized too. As shown in 5, this work could reduce the compression runtime of our UdeACompress around  $3\times-4\times$ . Heterogeneous hardware could be considered in both cases.

Compression of quality scores had an important impact in the compression ratio. Current highly specialized approaches should be considered for such task if higher compression ratios are to be achieved.

Finally, regarding the SIMD SAC algorithm, a higher degree of array element packing should be tested to see if it improves performance. Also, it should be investigated if major changes in the design of the sorting algorithm could make it more suitable to be efficiently combined with thread level parallelism. An important experiment is to compare if the drawbacks of a comparison based SIMD sorting (e.g. Quicksort or Mergesort) are hidden by the gain in performance that multi-threaded programming can bring.

# A. FASTQ Compression Results

**Table A-1.** Summary of performance results

T	SRR1282409			SRR3141946			DRR000604			SRR892505			SRR892403			SRR892407			
	CR	CT	DT	CR	CT	DT	CR	CT	DT	CR	CT	DT	CR	CT	DT	CR	CT	DT	
<b>FASTQ Compressors</b>																			
<b>Non-referential</b>																			
<b>Quip</b>	2	7.71	24.1	14.2	4.88	20.9	13.0	5.58	18.2	13.4	7.29	17.0	12.6	5.34	21.0	12.0	7.73	23.2	13.2
<b>SCALCE</b>	4	7.02	22.4	39.3	5.12	23.2	37.1	5.68	13.1	35.3	6.87	7.9	26.5	5.99	23.2	42.7	7.03	18.3	41.2
<b>DSRC 2.0</b>	24	4.13	<b>294</b>	<b>242</b>	3.84	<b>199</b>	<b>213</b>	4.37	<b>156</b>	<b>150</b>	3.99	<b>200</b>	<b>243</b>	3.93	<b>174</b>	<b>368</b>	4.0	<b>105</b>	<b>273</b>
<b>Fastqz</b>	4	<b>8.48</b>	2.9	4.5	5.39	2.5	4	5.86	4.3	3.6	<b>7.87</b>	2.6	3.4	6.83	2.7	3.9	<b>8.01</b>	1.9	3.2
<b>Referential</b>																			
<b>UdeAC</b>	1	7.29	2.8	10.9	<b>6.6</b>	3.0	11.5	<b>8</b>	2.7	11.8	6.8	1.5	11.1	<b>7.07</b>	4.7	11.7	7.3	4.6	11.1
<b>UdeACP</b>	24	-	9.9	-	-	10.8	-	-	13.1	-	-	22.4	-	-	19.7	-	-	18.0	-
<b>LWFQZIP 2</b>	10	6.76	4.0	9.9	4.99	4.4	10.0	6.43	10.2	14.4	6.98	6.7	7.2	6.36	8.3	7.0	6.91	7.6	9.7
<b>León</b>	24	4.62	21.2	52.4	4.30	19.1	48.6	4.69	15.8	28.8	5.38	15.7	29.7	5.26	23.8	52.9	4.21	27.0	57.0

T: Max number of used threads, CR: Compression Ratio, CT: Compression Throughput, DT: Decompression Throughput.

Bold numbers represent the respective maximum

UdeaC: UdeACompress

UdeACP: UdeACompress estimation of performances of a parallel version

UdeACMr: UdeACompress performance on mapped reads only

# B. FASTQ Read Sequences Compression

**Table B-1.** Summary of performance results

	SRR1282409			SRR3141946			DRR000604			SRR892505			SRR892403			SRR892407				
	T	CR	CT	DT	CR	CT	DT	CR	CT	DT	CR	CT	DT	CR	CT	DT	CR	CT	DT	
<b>FASTQ Compressors</b>																				
<b>Read sequences compressors</b>																				
<b>KPath</b>	10	<b>38</b>	1.2	2	<b>19</b>	0.9	1.7	-	-	-	57	1.7	2.4	31.4	1.9	2.4	70.9	2.6	2.2	
<b>ORCOM</b>	8	33.9	<b>26.6</b>	<b>216</b>	14.3	<b>27.2</b>	<b>153</b>	7.9	<b>13.8</b>	<b>121</b>	50.5	<b>21</b>	<b>325</b>	28	<b>24.6</b>	<b>120</b>	59.7	<b>29.1</b>	<b>192</b>	
<b>HARC</b>	8	33.4	6.8	58	13.4	7.5	46.7	6.8	4.5	26.8	<b>60.5</b>	15.3	141	<b>35.5</b>	12	62.6	<b>72.4</b>	18.8	74.5	
<b>Assembletrie</b>	8	-	-	-	-	-	-	-	-	-	-	-	-	22.4	5.8	46.5	33	4.8	48.6	
<b>UdeaC</b>	1	14.9	1.4	10.5	15.8	1.4	10.1	<b>14.9</b>	1.1	10.3	14.5	2.8	10.3	17.8	2.7	9.9	19.3	2.7	10.0	
<b>UdeaACP</b>	24	-	5.5	-	-	6.3	-	-	7.6	-	-	17.6	-	-	15.6	-	-	-	15.1	-
<b>UdeaCMr</b>	-	28.8	-	-	27.1	-	-	37.8	-	-	36.5	-	-	40.1	-	-	36.4	-	-	-

T: Max number of used threads, CR: Compression Ratio, CT: Compression Throughput, DT: Decompression Throughput.

Bold numbers represent the respective maximum

UdeaC: UdeaCCompress

UdeaACP: UdeaCCompress estimation of performances of a parallel version

UdeaCMr: UdeaCCompress performance on mapped reads only

# Bibliography

- [1] J. Lupski *et al.*, “Human genome at ten: The sequence explosion,” *Nature*, vol. 464, no. 7289, pp. 670–671, apr 2010. [Online]. Available: <https://doi.org/10.1038%2F464670a>
- [2] P.-R. Loh, M. Baym, and B. Berger, “Compressive genomics.” *Nature biotechnology*, vol. 30, no. 7, pp. 627–30, Jul. 2012. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/22781691>
- [3] K. Wetterstrand, “The cost of sequencing a human genome,” 2016, accessed: 2017-10-20. [Online]. Available: <https://www.genome.gov/sequencingcosts/>
- [4] S. Deorowicz and S. Grabowski, “Data compression for sequencing data,” *Algorithms for Molecular Biology*, vol. 8, no. 1, p. 25, 2013. [Online]. Available: <https://doi.org/10.1186%2F1748-7188-8-25>
- [5] RAID Incorporated, “Storing and Managing Petabytes of Genome Sequencing Data,” RAID Incorporated, Tech. Rep., 2015, [Online; accessed: 2015-03-23]. [Online]. Available: <http://webinfo.raidinc.com/storing-and-managing-petabytes-of-genome-sequencing-data>
- [6] L. D. Stein, “The case for cloud computing in genome informatics,” *Genome Biology*, vol. 11, no. 5, p. 207, 2010. [Online]. Available: <https://doi.org/10.1186%2Fgb-2010-11-5-207>
- [7] E. C. Hayden, “Is the \$1,000 genome for real?” *Nature*, jan 2014. [Online]. Available: <https://doi.org/10.1038%2Fnature.2014.14530>
- [8] K. Vanmechelen, J. Altmann, and O. F. Rana, Eds., *Economics of Grids, Clouds, Systems, and Services*. Springer Berlin Heidelberg, 2012. [Online]. Available: <https://doi.org/10.1007%2F978-3-642-28675-9>
- [9] “1000 Genomes. A Deep Catalog of Human Genetic Variation,” 2014, [Online; Accessed: 2014-10-03]. [Online]. Available: <http://www.1000genomes.org>
- [10] E. P. Consortium *et al.*, ““the encode (ENCyclopedia of DNA elements) project”,” *Science*, vol. 306, no. 5696, pp. 636–640, 2004.
- [11] G. England, “The 100,000 genomes project,” *The*, vol. 100, pp. 0–2, 2016.
- [12] N. Bakr and A. Sharawi, “Dna lossless compression algorithms: Review,” *American Journal of Bioinformatics Research*, vol. Vol. 3, pp. pp. 72–81, 11 2013.
- [13] Y. Zhang, K. Patel, T. Endrawis, A. Bowers, and Y. Sun, “A FASTQ compressor based on integer-mapped k-mer indexing for biologist,” *Gene*, vol. 579, no. 1, pp. 75–81, 2015. [Online]. Available: <http://dx.doi.org/10.1016/j.gene.2015.12.053>



- [14] “FASTQ Format specification,” 2014, [Online; Accessed: 2014-09-23]. [Online]. Available: <http://maq.sourceforge.net/fastq.shtml>
- [15] Andreas D. Baxeavanis and B. F. Ouellette Francis, *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins.*, 2nd ed., Wiley-Interscience, Ed. John Wiley & Sons, 2004.
- [16] S. Wandelt, M. Bux, and U. Leser, “Trends in Genome Compression,” *Current Bioinformatics*, pp. 1–24, 2013. [Online]. Available: <https://edit.rok.informatik.hu-berlin.de/wbi/research/publications/2013/2013-cbio.pdf>
- [17] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Transactions on Information Theory*, vol. I, no. 3, 1977.
- [18] ———, “Compression of individual sequences via variable-rate coding,” *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.
- [19] K. K. Kaipa, K. Lee, T. Ahn, and R. Narayanan, “System for random access dna sequence compression,” *IEEE International Conference on Bioinformatics and Biomedicine Workshops System*, pp. 853–854, 2010. [Online]. Available: [http://ieeexplore.ieee.org/xpls/abs\\_all.jsp?arnumber=5703942](http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5703942)
- [20] D. A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proceedings of the I.R.E.*, vol. 27, pp. 1098, 1101, 1952.
- [21] A. J. Pinho, D. Pratas, and S. P. Garcia, “GREn: A tool for efficient compression of genome resequencing data,” *Nucleic Acids Research*, vol. 40, no. 4, pp. 1–8, 2012.
- [22] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze, “Compression of next-generation sequencing reads aided by highly efficient de novo assembly,” *Nucleic Acids Research*, vol. 40, no. 22, pp. 1–9, 2012.
- [23] S. Christley, Y. Lu, C. Li, and X. Xie, “Human genomes as email attachments,” *Bioinformatics*, vol. 25, no. 2, pp. 274–275, 2009. [Online]. Available: <https://doi.org/10.1093/bioinformatics/btn582>
- [24] A. Guerra, J. Lotero, and S. Isaza, “Performance comparison of sequential and parallel compression applications for DNA raw data,” *The Journal of Supercomputing*, vol. 72, no. 12, pp. 4696–4717, jun 2016.
- [25] R. Giancarlo, D. Scaturro, and F. Utro, “Textual data compression in computational biology: a synopsis,” *Bioinformatics*, vol. 25, no. 13, pp. 1575–1586, feb 2009. [Online]. Available: <https://doi.org/10.1093%2Fbioinformatics%2Fbtp117>
- [26] T. Matsumoto, K. Sadakane, and H. Imai, “Biological sequence compression algorithms.” *Genome informatics. Workshop on Genome Informatics*, vol. 11, pp. 43–52, 2000.
- [27] A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone, “Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform.” *Bioinformatics (Oxford, England)*, vol. 28, no. 11, pp. 1415–9, Jun. 2012. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/22556365>

- [28] M. Burrows and D. A. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital Equipment Corporation, California, USA, Tech. Rep., 1994.
- [29] F. Hach, I. Numanagic, C. Alkan, S. C. Sahinalp, I. Numanagić, C. Alkan, and S. C. Sahinalp, "SCALCE: boosting sequence compression algorithms using locally consistent encoding." *Bioinformatics*, vol. 28, no. 23, pp. 3051–7, Dec. 2012. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3509486&tool=pmcentrez&rendertype=abstract>
- [30] X. Chen, S. Kwong, and M. Li, "A compression algorithm for DNA sequences," *IEEE Engineering in Medicine and Biology Magazine*, vol. 20, no. August, pp. 61–66, 2001. [Online]. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=940049>
- [31] W. Tembe, J. Lowey, E. Suh, T. Genomics, and N. Street, "G-SQZ: Compact encoding of genomic sequence and quality data," *Bioinformatics*, vol. 26, no. 17, pp. 2192–2194, 2010.
- [32] J. J. Selva and X. Chen, "SRComp: Short read sequence compression using burstersort and Elias omega coding," *PLoS ONE*, vol. 8, no. 12, pp. 1–7, 2013.
- [33] L. Janin, O. Schulz-Trieglaff, and A. J. Cox, "BEETL-fastq: a searchable compressed archive for DNA reads." *Bioinformatics (Oxford, England)*, pp. 1–6, 2014. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/24950811>
- [34] M. Howison, "High-throughput compression of FASTQ data with SeqDB," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 10, no. 1, pp. 213–218, 2013.
- [35] A. Dutta, M. M. Haque, T. Bose, C. Reddy, and S. S. Mande, "Fqc: A novel approach for efficient compression, archival, and dissemination of fastq datasets," *Journal of bioinformatics and computational biology*, vol. 13, no. 03, p. 1541003, 2015.
- [36] E. Grassi, F. D. Gregorio, and I. Molineris, "KungFQ: A simple and powerful approach to compress fastq files," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 6, pp. 1837–1842, 2012.
- [37] M. Nicolae, S. Pathak, and S. Rajasekaran, "LFQC: A lossless compression algorithm for FASTQ files," *Bioinformatics*, vol. 31, no. 20, pp. 3276–3281, 2015.
- [38] X. Zhan and D. Yao, "A novel method to compress high-throughput dna sequence read archive," in *Software Intelligence Technologies and Applications International Conference on Frontiers of Internet of Things 2014, International Conference on*, Dec 2014, pp. 58–61.
- [39] S. Deorowicz and S. Grabowski, "Compression of DNA sequence reads in FASTQ format." *Bioinformatics*, vol. 27, no. 6, pp. 860–862, Mar. 2011. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/21252073>
- [40] M. Hsi-Yang Fritz, R. Leinonen, G. Cochrane, E. Birney, M. H.-y. Fritz, R. Leinonen, G. Cochrane, and E. Birney, "Efficient storage of high throughput DNA sequencing data using reference-based compression." *Genome research*, vol. 21, no. 5, pp. 734–40, May 2011. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3083090&tool=pmcentrez&rendertype=abstract>

- [41] V. Yanovsky, "ReCoil - an algorithm for compression of extremely large datasets of dna data," *Algorithms for Molecular Biology*, vol. 6, no. 1, p. 23, Jan. 2011. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3219593&tool=pmcentrez&rendertype=abstract><http://www.almob.org/content/6/1/23>
- [42] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese, "Compressing genomic sequence fragments using SlimGene." *Journal of computational biology : a journal of computational molecular cell biology*, vol. 18, no. 3, pp. 401–13, Mar. 2011. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3123913&tool=pmcentrez&rendertype=abstract>
- [43] U.S. National Library of Medicine, "What is dna?" <https://ghr.nlm.nih.gov/chromosome>, 2017, accessed: 2017-10-20.
- [44] S. Isaza, "Multicore architectures for bioinformatics applications," Ph.D. dissertation, Delft University of Technology, 2011.
- [45] K. Daily, P. Rigor, S. Christley, X. Xie, and P. Baldi, "Data structures and compression algorithms for high-throughput sequencing technologies," *BMC Bioinformatics*, vol. 11, no. 1, p. 514, 2010. [Online]. Available: <https://doi.org/10.1186/1471-2105-11-514>
- [46] P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants," *Nucleic acids research*, vol. 38, no. 6, pp. 1767–1771, 2009.
- [47] M. C. Brandon, D. C. Wallace, and P. Baldi, "Data structures and compression algorithms for genomic sequence data," *Bioinformatics*, vol. 25, no. 14, pp. 1731–1738, may 2009. [Online]. Available: <https://doi.org/10.1093/Fbioinformatics/Fbtp319>
- [48] M. Nelson and J.-L. Gailly, *The Data Compression Book (2Nd Ed.)*. New York, NY, USA: MIS:Press, 1996.
- [49] Grumbach Stephane and F. Tahi, "A new challenge for compression algorithms : Genetic Sequences," *Information Processing & Management*, vol. 30, pp. 875–886, 1994. [Online]. Available: <https://hal.archives-ouvertes.fr/file/index/docid/180949/filename/grumbach.pdf>
- [50] J. K. Bonfield and M. V. Mahoney, "Compression of FASTQ and SAM Format Sequencing Data," *PLoS ONE*, vol. 8, no. 3, pp. 1–11, Jan. 2013. [Online]. Available: <http://dx.plos.org/10.1371/journal.pone.0059190>
- [51] L. Roguski and S. Deorowicz, "DSRC 2 - Industry oriented compression of FASTQ files," *Bioinformatics*, vol. 30, no. 15, pp. 2213–2215, 2014. [Online]. Available: <http://bioinformatics.oxfordjournals.org/content/30/15/2213>
- [52] R. Giancarlo, S. E. Rombo, and F. Utro, "Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies," *Briefings in Bioinformatics*, vol. 15, no. 3, pp. 390–406, 2014.

- [53] I. Numanagić, J. K. Bonfield, F. Hach, J. Voges, J. Ostermann, C. Alberti, M. Mattavelli, and S. C. Sahinalp, "Comparison of high-throughput sequencing data compression tools," *nature methods*, vol. 13, no. 12, p. 1005, 2016.
- [54] A. Mehta and B. Patel, "Dna compression using hash based data structure," *International Journal of Information Technology and Knowledge Management*, pp. 383–386, 01 2010.
- [55] G. Vey, "Differential direct coding: a compression algorithm for nucleotide sequence data." *Database : the journal of biological databases and curation*, vol. 2009, p. bap013, Jan. 2009. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2797453&tool=pmcentrez&rendertype=abstract>
- [56] R. K. Bharti and R. K. Singh, "A Biological Sequence Compression based on Look up Table (LUT) using Complementary Palindrome of Fixed Size," *International Journal of Computer Applications*, vol. 35, no. 11, pp. 55–58, 2011.
- [57] P. R. Rajeswari, A. Apparao, and V. Kumar, "Genbit compress tool (gbc): A java-based tool to compress dna sequences and compute compression ratio (bits/base) of genomes," *arXiv preprint arXiv:1006.1193*, 2010.
- [58] P. Rajarajeswari and A. Apparao, "Normalized Distance Matrix Method for Construction of Phylogenetic Trees Using New Compressor - Dnabit Compress . I : Introduction," *Bioinformatics*, vol. 2, no. 1, pp. 89–97, 2011.
- [59] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel, "Iterative dictionary construction for compression of large DNA data sets," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 9, no. 1, pp. 137–149, jan 2012. [Online]. Available: <https://doi.org/10.1109%2Ftcb.2011.82>
- [60] K. K. Kaipa, A. S. Bopardikar, S. Abhilash, P. Venkataraman, K. Lee, T. Ahn, and R. Narayanan, "Algorithm for DNA sequence compression based on prediction of mismatch bases and repeat location," *2010 IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW)*, dec 2010. [Online]. Available: <https://doi.org/10.1109%2Fbibmw.2010.5703941>
- [61] S. Golomb, "Run-length encodings," *IEEE Trans. Inf. Theory*, vol. 12, pp. 399–401, 1965.
- [62] P. Elias, "Universal codeword sets and representations of the integers." *IEEE Trans. Inf. Theory*, vol. 21, pp. 194–203, 1975.
- [63] R. L. Graham, D. E. Knuth, O. Patashnik, and S. Liu, "Concrete mathematics: a foundation for computer science," *Computers in Physics*, vol. 3, no. 5, pp. 106–107, 1989.
- [64] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.
- [65] M. D. Cao, T. I. Dix, L. Allison, and C. Mears, "A simple statistical algorithm for biological sequence compression," *2007 Data Compression Conference (DCC07)*, 2007. [Online]. Available: <https://doi.org/10.1109%2Fdcc.2007.7>

- [66] I. Tabus and G. Korodi, "Genome compression using normalized maximum likelihood models for constrained markov sources," *2008 IEEE Information Theory Workshop*, may 2008. [Online]. Available: <https://doi.org/10.1109%2Fitw.2008.4578663>
- [67] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino, "An Extension of the Burrows Wheeler Transform and Applications to Sequence Comparison and Data Compression," *CPM*, vol. 3537 of LN, pp. 178–189, 2005.
- [68] M. J. Bauer, A. J. Cox, and G. Rosone, "Lightweight algorithms for constructing and inverting the BWT of string collections," *Theoretical Computer Science*, vol. 483, pp. 134–148, Apr. 2013. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0304397512001211>
- [69] G. Benoit, C. Lemaitre, D. Lavenier, and G. Rizk, "Compression of high throughput sequencing data with probabilistic de Bruijn graph," *arXiv preprint arXiv:1412.5932*, 2014. [Online]. Available: <http://arxiv.org/abs/1412.5932>
- [70] C. Kingsford and R. Patro, "Reference-based compression of short-read sequences using path encoding," *Bioinformatics*, vol. 31, no. 12, pp. 1920–1928, 2015. [Online]. Available: <http://bioinformatics.oxfordjournals.org/lookup/doi/10.1093/bioinformatics/btv071>
- [71] M. Chlopkowski, M. Antczak, M. Slusarczyk, A. Wdowinski, M. Zajackowski, and M. Kasprzak, "High-order statistical compressor for long-term storage of DNA sequencing data," *RAIRO - Operations Research*, vol. 50, no. 2, pp. 351–361, 2015. [Online]. Available: <http://www.rairo-ro.org/articles/ro/pdf/forth/ro150039-s.pdf>
- [72] S. Grabowski and S. Deorowicz, "Engineering Relative Compression of Genomes," *CoRR*, vol. abs/1103.2, Mar. 2011. [Online]. Available: <http://arxiv.org/abs/1103.2351v1>
- [73] F. Gebali, *Algorithms and parallel computing*. John Wiley & Sons, 2011, vol. 84.
- [74] S. Grabowski, S. Deorowicz, and L. Roguski, "Disk-based compression of data from genome sequencing," *Bioinformatics*, vol. 31, no. 9, pp. 1389–1395, 2015.
- [75] Z. A. Huang, Z. Wen, Q. Deng, Y. Chu, Y. Sun, and Z. Zhu, "LW-FQZip 2: A parallelized reference-based compression of FASTQ files," *BMC Bioinformatics*, vol. 18, no. 1, pp. 1–8, 2017.
- [76] C. Xavier and S. Iyengar, *Introduction to Parallel Algorithms*, ser. A Wiley interscience publication. Wiley, 1998.
- [77] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.
- [78] F. Sanchez, "Exploiting multiple levels of parallelism in bioinformatics applications," Ph.D. dissertation, PhD thesis, Technical University of Catalonia, 2011.
- [79] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero, "Vsr sort: A novel vectorised sorting algorithm & architecture extensions for future microprocessors," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 26–38.

- [80] T. Hayes, O. Palomar, and M. Valero, “Novel vector architecture for data management,” Ph.D. dissertation, PhD thesis, Technical University of Catalonia, 2016.
- [81] Intel, “Intel Architecture Instruction Set Extensions Programming Reference,” *Technology*, no. February, 2012.
- [82] C. Lomont, “Introduction to intel advanced vector extensions,” *Intel White Paper*, pp. 1–21, 2011.
- [83] V. Bhola, A. S. Bopardikar, R. Narayanan, K. Lee, and T. Ahn, “No-reference compression of genomic data stored in FASTQ format,” *Proceedings - 2011 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2011*, pp. 147–150, 2011.
- [84] M. H. Mohammed, A. Dutta, T. Bose, S. Chadaram, and S. S. Mande, “DELIMINATE—a fast and efficient method for loss-less compression of genomic sequences: sequence analysis.” *Bioinformatics (Oxford, England)*, vol. 28, no. 19, pp. 2527–9, 2012. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/22833526>
- [85] Z. Zhu, Y. Zhang, Z. Ji, S. He, and X. Yang, “High-throughput DNA sequence data compression.” *Briefings in bioinformatics*, vol. 16, no. 1, 2013. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/24300111>
- [86] M. Sardaraz, M. Tahir, and A. A. Ikram, “Advances in high throughput dna sequence data compression,” *Journal of Bioinformatics and Computational Biology*, vol. 0, no. 0, p. 1630002, 0, pMID: 26846812. [Online]. Available: <http://www.worldscientific.com/doi/abs/10.1142/S0219720016300021>
- [87] X. Chen, S. Kwong, and M. Li, “A Compression Algorithm for DNA Sequences and Its Applications in Genome Comparison.” *Genome informatics. Workshop on Genome Informatics*, vol. 10, pp. 51–61, Jan. 1999. [Online]. Available: <http://www.ncbi.nlm.nih.gov/pubmed/11072342>
- [88] M. Oberhumer, “Lzo real-time data compression library,” Feb. 2015, [Online; Accessed: 2016-03-03]. [Online]. Available: <http://www.oberhumer.com/opensource/lzo/>
- [89] I. Pavlov, “7-zip,” Feb. 2016, [Online; Accessed: 2016-03-03]. [Online]. Available: <http://www.7-zip.org>
- [90] “WinRAR archiver, a powerful tool to process RAR and ZIP files.” [Online]. Available: <http://www.rarlab.com/>
- [91] M. Adler, “PIGZ Documentation,” [Online; accessed: 2014-12-03]. [Online]. Available: <http://zlib.net/pigz/pigz.pdf>
- [92] Linux man page, “Pbzip2: Parallel bzip2 file compressor,” 2014, [Online; accessed: 2014-12-03]. [Online]. Available: <http://linux.die.net/man/1/pbzip2>
- [93] M. N. Sakib, J. Tang, W. J. Zheng, and C. T. Huang, “Improving transmission efficiency of large sequence alignment/map (SAM) files,” *PLoS ONE*, vol. 6, no. 12, pp. 2–5, 2011.

- [94] G. Guo, S. Qiu, Z. Ye, B. Wang, L. Fang, M. Lu, S. See, and R. Mao, "GPU-Accelerated Adaptive Compression Framework for Genomics Data," *2013 IEEE International Conference on Big Data GPU-Accelerated*, pp. 181–186, 2013.
- [95] F. Awan and A. Mukherjee, *Lossless Compression Handbook*, ser. Communications, Networking and Multimedia. Elsevier Science, 2002, ch. Text Compression, pp. 227–245.
- [96] A. Carus and A. Mesut, "Fast Text Compression Using Multiple Static Dictionaries," *Information Technology Journal*, vol. 9, pp. 1013–1021, 2010.
- [97] M. Crochemore and T. Lecroq, *The Computer Science and Engineering Handbook*. CRC Press, 2012, ch. Pattern matching and text compression algorithms, pp. 3–77.
- [98] "7-Zip Sourceforge Editor's Review," Feb. 2016, [Online; Accessed: 2016-03-03]. [Online]. Available: <https://sourceforge.net/projects/sevenzzip/editorial/?source=psp>
- [99] M. Mahoney, "Data compression explained," Feb. 2016, [Online; Accessed: 2016-03-03]. [Online]. Available: <http://mattmahoney.net/dc/dce.html#Section.523>
- [100] M. V. Mahoney, "Adaptive weighing of context models for lossless data compression," *Florida Tech., Melbourne, USA*, vol. CS-2005-16, no. x, pp. 1–6, 2005. [Online]. Available: <http://professor.unisinos.br/linds/teoinfo/paq.pdf>
- [101] D. Salomon, D. Bryant, and G. Motta, *Handbook of Data Compression*. Springer London, 2010.
- [102] T. Batu, F. Ergun, and C. Sahinalp, "Oblivious string embeddings and edit distance approximations," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. Society for Industrial and Applied Mathematics, 2006, pp. 792–801.
- [103] Biocancer Research Journal, "Transcriptoma," 2014, [Online; Accessed: 2014-12-03]. [Online]. Available: <http://www.biocancer.com/journal/1353/31-transcriptoma>
- [104] J. Pevsner, *Bioinformatics and Functional Genomics, 2nd Ed*. Springer Verlag, 2009.
- [105] S. Deorowicz and S. Grabowski, "Robust relative compression of genomes with random access," *Bioinformatics*, vol. 27, no. 21, pp. 2979–2986, 2011.
- [106] D. S. Pavlichin, T. Weissman, and G. Yona, "The human genome contracts again," *Bioinformatics*, vol. 29, no. 17, pp. 2199–2202, 2013.
- [107] C. Wang and D. Zhang, "A novel compression tool for efficient storage of genome resequencing data," *Nucleic Acids Research*, vol. 39, no. 7, pp. 5–10, Apr. 2011. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3074166&tool=pmcentrez&rendertype=abstract>
- [108] B. G. Chern, I. Ochoa, a. Manolakos, a. No, K. Venkat, and T. Weissman, "Reference based genome compression," *2012 IEEE Information Theory Workshop, ITW 2012*, pp. 427–431, 2012.

- [109] S. Wandelt and U. Leser, "Adaptive efficient compression of genomes." *Algorithms for Molecular Biology*, vol. 7, no. 1, pp. 1–9, 2012. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3541066&tool=pmcentrez&rendertype=abstract><http://www.biomedcentral.com/content/pdf/1748-7188-7-30.pdf>
- [110] Z. Zhu, J. Zhou, Z. Ji, and Y.-H. Shi, "DNA Sequence Compression Using Adaptive Particle Swarm Optimization-Based Memetic Algorithm," *IEEE Transactions on Evolutionary Computation*, vol. 15, no. 5, pp. 643–658, 2011.
- [111] K. Mishra, D. Anupam Aagarwal, E. Abdelhadi, and P. Srivastava, "An efficient horizontal and vertical method for online dna sequence compression," *International Journal of Computer Applications*, vol. 3, 06 2010.
- [112] S. Kuruppu, S. Puglisi, and J. Zobel, "Optimized relative Lempel-Ziv compression of genomes," *Proceedings of the Thirty-Fourth . . .*, 2011. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2459307>
- [113] S. Wandelt and U. Leser, "Mrcsi: Compressing and searching string collections with multiple references," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 461–472, 2015.
- [114] H. Afify, M. Islam, M. A. Wahed, and Y. M. Kadah, "Genomic Sequences Differential Compression Model Academy of Scientific Research and Technology," *Radio Science*, no. March, pp. 1–7, 2010.
- [115] S. Wandelt and U. Leser, "FRESCO: Referential compression of highly similar sequences," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 10, no. 5, pp. 1275–1288, 2013.
- [116] S. Deorowicz, A. Danek, and M. Niemiec, "GDC 2: Compression of large collections of genomes," *Scientific Reports*, vol. 5, no. 1, jun 2015. [Online]. Available: <https://doi.org/10.1038/srep11565>
- [117] S. Saha and S. Rajasekaran, "ERGC: An efficient referential genome compression algorithm," *Bioinformatics*, vol. 31, no. 21, pp. 3468–3475, 2014.
- [118] T. Sam, B. A. M. Format, and S. Working, "Sequence Alignment / Map Format Specification," *The SAM/BAM Format Specification Working Group*, no. May, pp. 1–16, 2015.
- [119] P. Li, X. Jiang, S. Wang, J. Kim, H. Xiong, and L. Ohno-Machado, "HUGO: Hierarchical mUlti-reference Genome cOmpression for aligned reads." *Journal of the American Medical Informatics Association : JAMIA*, vol. 21, no. 2, pp. 363–73, 2014. [Online]. Available: <http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3932469&tool=pmcentrez&rendertype=abstract>
- [120] F. Hach, I. Numanagic, and S. C. Sahinalp, "Deez: reference-based compression by local assembly," *Nature methods*, vol. 11, no. 11, p. 1082, 2014.
- [121] N. Popitsch and A. Von Haeseler, "NGC: Lossless and lossy compression of aligned high-throughput sequencing data," *Nucleic Acids Research*, vol. 41, no. 1, pp. 1–12, 2013.
- [122] F. Campagne, K. C. Dorff, N. Chambwe, J. T. Robinson, and J. P. Mesirov, "Compression of structured high-throughput sequencing data," *PLoS ONE*, vol. 8, no. 11, 2013.



- [123] “CRAM format specification (version 3.0),” Apr. 2018, [Online; Accessed: 2018-04-10]. [Online]. Available: <https://samtools.github.io/hts-specs/CRAMv3.pdf>
- [124] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin, “The Sequence Alignment/Map format and SAMtools,” *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.
- [125] Z. Zhu, L. Li, Y. Zhang, Y. Yang, and X. Yang, “Comp Map: A reference-based compression program to speed up read mapping to related reference sequences,” *Bioinformatics*, vol. 31, no. 3, pp. 426–428, 2014.
- [126] Y. Zhang, L. Li, Y. Yang, X. Yang, S. He, and Z. Zhu, “Light-weight reference-based compression of FASTQ data.” *BMC bioinformatics*, vol. 16, no. 1, p. 188, 2015. [Online]. Available: <http://www.biomedcentral.com/1471-2105/16/188>{%}5Cn<http://www.ncbi.nlm.nih.gov/pubmed/26051252>{%}5Cn<http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=PMC4459677>
- [127] Ł. Roguski and P. Ribeca, “Cargo: effective format-free compressed storage of genomic information,” *Nucleic acids research*, vol. 44, no. 12, pp. e114–e114, 2016.
- [128] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, “KMC 2: Fast and resource-frugal k-mer counting,” *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2014.
- [129] T. Braquelaire, M. Gasparoux, M. Raffinot, and R. Uricaru, “On the shortest common superstring of NGS reads,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10185 LNCS, pp. 97–111, 2017.
- [130] F. Claude, A. Farina, M. A. Martínez-Prieto, and G. Navarro, “Compressed q-gram indexing for highly repetitive biological sequences,” in *Bioinformatics and BioEngineering (BIBE), 2010 IEEE International Conference on*. IEEE, 2010, pp. 86–91.
- [131] M. Kokot, M. Długosz, and S. Deorowicz, “KMC 3: counting and manipulating k-mer statistics,” *Bioinformatics*, vol. 33, no. 17, pp. 2759–2761, 2017.
- [132] R. J. Urbanowicz, R. S. Olson, P. Schmitt, M. Meeker, and J. H. Moore, “Benchmarking relief-based feature selection methods for bioinformatics data mining,” *Journal of biomedical informatics*, 2018.
- [133] C. S. Greene, N. M. Penrod, J. Kiralis, and J. H. Moore, “Spatially uniform relief (surf) for computationally-efficient filtering of gene-gene interactions,” *BioData mining*, vol. 2, no. 1, p. 5, 2009.
- [134] B. A. McKinney, B. C. White, D. E. Grill, P. W. Li, R. B. Kennedy, G. A. Poland, and A. L. Oberg, “Reliefseq: a gene-wise adaptive-k nearest-neighbor feature selection tool for finding gene-gene interactions and main effects in mrna-seq gene expression data,” *PloS one*, vol. 8, no. 12, p. e81527, 2013.
- [135] M. E. Stokes and S. Visweswaran, “Application of a spatially-weighted relief algorithm for ranking genetic predictors of disease,” *BioData mining*, vol. 5, no. 1, p. 20, 2012.

- [136] Y. Sun, S. Todorovic, and S. Goodison, "Local-learning-based feature selection for high-dimensional data analysis," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 9, pp. 1610–1626, 2010.
- [137] Y. Sun, "Iterative relief for feature weighting: algorithms, theories, and applications," *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, 2007.
- [138] J. H. Moore and B. C. White, "Tuning relief for genome-wide genetic analysis," in *European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*. Springer, 2007, pp. 166–175.
- [139] S. Vinga and J. Almeida, "Alignment-free sequence comparison—a review," *Bioinformatics*, vol. 19, no. 4, pp. 513–523, 2003.
- [140] I. Borozan, S. Watt, and V. Ferretti, "Integrating alignment-based and alignment-free sequence similarity measures for biological sequence classification," *Bioinformatics*, vol. 31, no. 9, pp. 1396–1404, 2015. [Online]. Available: <http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/btv006>
- [141] I. Alsmadi and M. Nuser, "String matching evaluation methods for dna comparison," *International Journal of Advanced Science and Technology*, vol. 47, no. 1, pp. p13–32, 2012.
- [142] Z. Xing, J. Pei, and E. Keogh, "A brief survey on sequence classification," *ACM Sigkdd Explorations Newsletter*, vol. 12, no. 1, pp. 40–48, 2010.
- [143] S. Mantaci, A. Restivo, and M. Sciortino, "Distance measures for biological sequences: Some recent approaches," *International Journal of Approximate Reasoning*, vol. 47, no. 1, pp. 109–124, 2008. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0888613X07000382>
- [144] Y. Saeys, I. Inza, and P. Larranaga, "A review of feature selection techniques in bioinformatics," *Bioinformatics*, vol. 23, no. 19, pp. 2507–2517, 2007. [Online]. Available: <http://bioinformatics.oxfordjournals.org/cgi/doi/10.1093/bioinformatics/btm344>
- [145] X. Chen and J. Chen, "Emerging Patterns and Classification Algorithms for DNA Sequence," *Journal of Software*, vol. 6, no. 6, pp. 985–992, 2011. [Online]. Available: <http://ojs.academypublisher.com/index.php/jsw/article/view/4226>
- [146] A. Firth, T. Bell, A. Mukherjee, and D. Adjeroh, "A comparison of bwt approaches to string pattern matching," *Software: Practice and Experience*, vol. 35, no. 13, pp. 1217–1258, 2005.
- [147] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Machine learning*, vol. 46, no. 1-3, pp. 389–422, 2002.
- [148] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 42, no. 2, pp. 513–529, 2012.

- [149] A. W.-C. Liew, H. Yan, and M. Yang, "Pattern recognition techniques for the emerging field of bioinformatics: A review," *Pattern Recognition*, vol. 38, no. 11, pp. 2055–2073, 2005. [Online]. Available: <http://linkinghub.elsevier.com/retrieve/pii/S0031320305001263>
- [150] D. de Ridder, J. de Ridder, and M. J. Reinders, "Pattern recognition in bioinformatics," *Briefings in bioinformatics*, vol. 14, no. 5, pp. 633–647, 2013.
- [151] G. L. Rosen, E. R. Reichenberger, and A. M. Rosenfeld, "Nbc: the naive bayes classification tool webserver for taxonomic classification of metagenomic reads," *Bioinformatics*, vol. 27, no. 1, pp. 127–129, 2010.
- [152] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "An empirical study of delta algorithms," in *International Workshop on Software Configuration Management*. Springer, 1996, pp. 49–66.
- [153] J. Fu and S. Dong, "All-cqs: Adaptive locality-based lossy compression of quality scores," in *Bioinformatics and Biomedicine (BIBM), 2017 IEEE International Conference on*. IEEE, 2017, pp. 353–359.
- [154] J. Voges, J. Ostermann, and M. Hernaez, "Calq: compression of quality values of aligned sequencing data," *Bioinformatics*, p. btx737, 2017.
- [155] C. Rockmann, C. Endrullat, M. Frohme, and H. Pospisil, "Next generation sequence analysis," in *Reference Module in Life Sciences*. Elsevier, 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/B9780128096338202069>
- [156] Imperial College London, "Read alignment," <http://www.imperial.ac.uk/bioinformatics-data-science-group/resources/software/next-generation-sequencing-ngs-software/read-alignment/>, 2017, accessed: 2017-10-23.
- [157] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.
- [158] P. Ferragina, J. Sirén, and R. Venturini, "Distribution-aware compressed full-text indexes," *Algorithmica*, vol. 67, no. 4, pp. 529–546, apr 2013. [Online]. Available: <https://doi.org/10.1007%2Fs00453-013-9782-3>
- [159] S. Grabowski, M. Raniszewski, and S. Deorowicz, "Fm-index for dummies," in *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation*, S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, Eds. Springer International Publishing, 2017, pp. 189–201.
- [160] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, mar 1970. [Online]. Available: <https://doi.org/10.1016%2F0022-2836%2870%2990057-4>
- [161] S. Gog, J. Bader, T. Beller, and M. Petri, "Sdsl - succinct data structure library," <https://github.com/simongog/sdsl-lite>, 2017, accessed: 2017-10-22.

- [162] J. A. Lotero García, “Improving genomic data compression through parallel computing,” Master’s thesis, Universidad de Antioquia, Colombia, 2018.
- [163] J. Lotero, A. Benavides, A. Guerra, and S. Isaza, “Udealignc: Fast alignment for the compression of dna reads,” in *2018 IEEE Colombian Conference on Communications and Computing (COLCOM)*. IEEE, 2018, pp. 1–6.
- [164] H. Wang’s, “A faster openmp radix sort implementation,” 2017, accessed: 2017-10-22. [Online]. Available: <https://haichuanwang.wordpress.com/tag/algorithm/>
- [165] D. Padua, *Encyclopedia of parallel computing*. Springer Science & Business Media, 2011.
- [166] R. Sinha, J. Zobel, and D. Ring, “Cache-efficient string sorting using copying,” *Journal of Experimental Algorithmics (JEA)*, vol. 11, pp. 1–2, 2007.
- [167] V. Eijkhout, *Introduction to High Performance Scientific Computing*. The University of Texas at Austin, 2014.
- [168] W. Li and J. Freudenberg, “Mappability and read length,” *Frontiers in Genetics*, vol. 5, no. NOV, pp. 1–1, 2014.
- [169] M. O. Pollard, D. Gurdasani, A. J. Mentzer, T. Porter, and M. S. Sandhu, “Long reads: their purpose and place,” *Human Molecular Genetics*, 2018.
- [170] A. Guerra, J. Lotero, J. É. Aedo, and S. Isaza, “Tackling the challenges of fastq referential compression,” *Bioinformatics and biology insights*, vol. 13, p. 1177932218821373, 2019.
- [171] S. Chandak, K. Tatwawadi, and T. Weissman, “Compression of genomic sequencing reads via hash-based reordering: algorithm and analysis,” *Bioinformatics*, vol. 34, no. 4, pp. 558–567, 2017.
- [172] A. A. Ginart, J. Hui, K. Zhu, I. Numanagić, T. A. Courtade, S. C. Sahinalp, and N. T. David, “Optimal compressed representation of high throughput sequence data via light assembly,” *Nature communications*, vol. 9, no. 1, p. 566, 2018.
- [173] D. Sims, I. Sudbery, N. E. Illott, A. Heger, and C. P. Ponting, “Sequencing depth and coverage: key considerations in genomic analyses,” *Nature Reviews Genetics*, vol. 15, no. 2, p. 121, 2014.
- [174] J. C. Na, H. Kim, H. Park, T. Lecroq, M. Léonard, L. Mouchard, and K. Park, “Fm-index of alignment: A compressed index for similar strings,” *Theoretical Computer Science*, vol. 638, pp. 159 – 170, 2016, pattern Matching, Text Data Structures and Compression. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0304397515007288>
- [175] J. C. Na, H. Kim, S. Min, H. Park, T. Lecroq, M. Léonard, L. Mouchard, and K. Park, “Fm-index of alignment with gaps,” *Theoretical Computer Science*, vol. 710, pp. 148 – 157, 2018, advances in Algorithms and Combinatorics on Strings (Honoring 60th birthday for Prof. Costas S. Iliopoulos). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0304397517301536>
- [176] H. Cheng, M. Wu, and Y. Xu, “Fmtree: A fast locating algorithm of fm-indexes for genomic data,” *Bioinformatics*, vol. 34, no. 3, pp. 416–424, 2017.

- [177] B. Langmead, “Introduction to the Burrows-Wheeler Transform and FM Index,” *JHU’s*, pp. 1–12, 2013.
- [178] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, “A taxonomy of suffix array construction algorithms,” *acm Computing Surveys (CSUR)*, vol. 39, no. 2, p. 4, 2007.
- [179] J. Sirén, “Burrows-wheeler transform for terabases,” in *Data Compression Conference (DCC), 2016*. IEEE, 2016, pp. 211–220.
- [180] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53 – 86, 2004, the 9th International Symposium on String Processing and Information Retrieval.
- [181] M. A. Maniscalco and S. J. Puglisi, “Faster lightweight suffix array construction,” in *Proc. of International Workshop On Combinatorial Algorithms (IWOCA)*. Citeseer, 2006, pp. 16–29.
- [182] J. Labeit, J. Shun, and G. E. Blelloch, “Parallel lightweight wavelet tree, suffix array and fm-index construction,” *Journal of Discrete Algorithms*, vol. 43, pp. 2–17, 2017.
- [183] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, “Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 351–362.
- [184] —, “Fast sort on cpus, gpus and intel mic architectures,” *Intel Labs*, pp. 77–80, 2010.
- [185] P. Helluy, ““A portable implementation of the radix sort algorithm in OpenCL,”” May 2011, working paper or preprint. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-00596730>
- [186] M. Kokot, S. Deorowicz, and A. Debudaj-Grabysz, “Sorting data on ultra-large scale with raduls,” in *International Conference: Beyond Databases, Architectures and Structures*. Springer, 2017, pp. 235–245.
- [187] M. Kokot, S. Deorowicz, and M. Długosz, “Even faster sorting of (not only) integers,” in *International Conference on Man–Machine Interactions*. Springer, 2017, pp. 481–491.
- [188] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, “Paradis: an efficient parallel algorithm for in-place radix sort,” *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1518–1529, 2015.
- [189] D. Lemire, L. Boytsov, and N. Kurz, “Simd compression and the intersection of sorted integers,” *Software: Practice and Experience*, vol. 46, no. 6, pp. 723–749, 2016.
- [190] H. Inoue and K. Taura, “Simd-and cache-friendly algorithm for sorting an array of structures,” *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1274–1285, 2015.
- [191] M. Zagha and G. E. Blelloch, “Radix sort for vector multiprocessors,” in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 1991, pp. 712–721.

- [192] S.-J. Lee, M. Jeon, D. Kim, and A. Sohn, "Partitioned parallel radix sort," *Journal of Parallel and Distributed Computing*, vol. 62, no. 4, pp. 656–668, 2002.
- [193] N. J. Larsson and K. Sadakane, *Faster suffix sorting*. Citeseer, 1999.
- [194] C. Nugteren, G.-J. van den Braak, H. Corporaal, and B. Mesman, "High performance predictable histogramming on gpus: exploring and evaluating algorithm trade-offs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 1.
- [195] R. Shams, R. Kennedy *et al.*, "Efficient histogram algorithms for nvidia cuda compatible devices," in *Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS)*. Citeseer, 2007, pp. 418–422.
- [196] V. Podlozhnyuk, "Histogram calculation in cuda," *NVIDIA white paper*, 2007.
- [197] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking simd vectorization for in-memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1493–1508.
- [198] A. Shahbahrani, B. Juurlink, and S. Vassiliadis, "Simd vectorization of histogram functions," in *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*. IEEE, 2007, pp. 174–179.
- [199] A. Shahbahrani, J. Y. Hur, B. Juurlink, and S. Wong, "Fpga implementation of parallel histogram computation," in *2nd HiPEAC Workshop on Reconfigurable Computing*. Published, 2008, pp. 63–72.
- [200] W. Jung, J. Park, and J. Lee, "Versatile and scalable parallel histogram construction," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 127–138.
- [201] H. Wang, "A faster openmp radix sort implementation," <https://haichuanwang.wordpress.com/2014/05/26/a-faster-openmp-radix-sort-implementation/>, 2014, accessed: 2017-07-20.

# Signatures

We endorse the submission of the modified version of the PhD thesis,

**Efficient Storage of Genomic Sequences in High Performance Computing Systems**

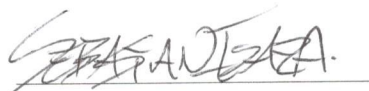
which has been corrected according to minor modifications indicated by the evaluators.



**Aníbal J. Guerra Soler**  
PhD Student



**José Edison Aedo Cobo**  
Advisor



**Sebastián Isaza Ramírez**  
Advisor