BOOSTING HIGH THROUGHPUT SEQUENCING DATA COMPRESSION ALGORITHMS USING REORDERING

by

Ibrahim Numanagić

B.Sc., University of Sarajevo, Bosnia and Herzegovina, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Ibrahim Numanagić 2013
SIMON FRASER UNIVERSITY
Spring 2013

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

APPROVAL

Name: Ibrahim Numanagić

Degree: Master of Science

Title of Thesis: Boosting High Throughput Sequencing Data Compression Algorithms Using Reordering

Examining Committee: Mr. Brad Bart
Chair

Dr. Süleyman Cenk Şahinalp, Senior Supervisor

Dr. Ayşe Funda Ergün, Supervisor

Dr. Martin Ester, SFU Examiner

Date Approved: April 19, 2013

Partial Copyright Licence

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website (www.lib.sfu.ca) at http://summit/sfu.ca and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

While licensing SFU to permit the above uses, the author retains copyright in the thesis, project or extended essays, including the right to change the work for subsequent purposes, including editing and publishing the work in whole or in part, and licensing other parties, as the author may desire.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, British Columbia, Canada

revised Fall 2011

Abstract

The high throughput sequencing (HTS) platforms generate unprecedented amounts of data that introduce challenges for the computational infrastructure. Currently, most HTS data is compressed through general purpose algorithms such as gzip. These algorithms are not designed for compressing data generated by the HTS platform, as they do not take advantage of the specific nature of genomic sequence data.

Here we present SCALCE, a "boosting" scheme based on Locally Consistent Parsing technique which reorganizes the reads in a way that results in a higher compression speed and compression rate, independent of the compression algorithm in use and without using a reference genome. Our tests indicate that SCALCE improves compression rate and time of gzip significantly. We also showed that reordering problem can be considered as an instance of set-cover problem, and that Locally Consistent Parsing is practically good as the best known approximation of set-cover problem.

keywords: FASTQ, Genome Sequence Compression, High Throughput Sequencing Technology, Lempel-Ziv Techniques, Locally Consistent Parsing, Boosting

To my beloved parents Hazim and Šahza,
and my dear siblings Ishak, Kerima and Sadik

İlim ilim bilmektir
İlim kendin bilmektir
Sen kendini bilmezsin
Ya nice okumaktır

Knowledge should mean a full grasp of knowledge:
Knowledge means to know yourself, heart and soul.
If you have failed to understand yourself,
Then all of your reading has missed its call.

— Hz. Yunus Emre

Acknowledgements

First of all, I would like to thank my senior supervisor, Dr. S. Cenk Şahinalp, for his extensive support, patience and guidance during my studies, and for the opportunity to work in his exceptional laboratory.

I would specially like to thank Faraz Hach for all his guidance and support during my studies, and for conceiving and leading the project which was used as a basis of this thesis, since this work would not be possible without his help.

Also, I would like to thank dr. Can Alkan for the writing help, software ideas and improvements, and experimental results. I would like to thank MITACS Accelerate Internship programme for providing me an opportunity to work on this project in a industrial environment. In addition, I would like to thank the fellows of the SFU Computational Biology lab for all their help and support.

I would like to thank Jianqiao Li for the provided moral support during the writing process, and Narek Nalbandyan for support and many fruitful conversations and advices.

Last but not the least, I am thankful to my wonderful family for their unlimited support during my studies. That is why I dedicate this thesis to them.

Contents

Approval ii

Partial Copyright License iii

Abstract iv

Dedication v

Quotation vi

Acknowledgements vii

Contents viii

List of Tables x

List of Figures xi

1 Introduction 1

2 Preliminaries 4

2.1 High-throughput sequencing 4

2.2 LZ77 family of compression algorithms 5

2.3 A theoretical exposition to the LCP technique. 5

2.4 Arithmetic encoding 9

2.5 FASTQ files 10

3 Methods **12**
3.1 Problem definition 12
3.2 Set cover approach for identifying the core substrings 13
3.3 A practical implementation of LCP for reordering reads. 14
3.4 A data structure for identifying core substrings of reads. 16
3.5 Compressing the quality scores. 16

4 Results **21**
4.1 Lossy compression effects 25
4.2 Set cover approach 26

5 Conclusion **28**
5.1 Future work 29

Bibliography **30**

Index **33**

List of Tables

4.1 Input data statistics and compression rates achieved by gzip only and SCALCE + gzip on reads from the P. aeruginosa RNA-Seq library. File sizes are reported in megabytes. . . . 22
4.2 Input data statistics and compression rates achieved by gzip only and SCALCE + gzip + AC on complete FASTQ files. File sizes are reported in megabytes. 23
4.3 Run time for running gzip alone and SCALCE + gzip + AC on complete FASTQ files. . . . 24
4.4 Comparison of single-threaded SCALCE with DSRC. DSRC was tested using the `-l` option except on the WGS sample, where it crashed. Instead we had to use a faster but less powerful setting for this data set. . . . 24
4.5 Comparison of single-threaded SCALCE with BEETL. Here, the data sets contained only reads from the FASTQ file, as BEETL supports only FASTA file format. . . . 25
4.6 Number of SNPs found in the NA18507 genome using original qualities and transformed qualities with 30% noise reduction. Also reported are the number and percentage of novel SNPs in regions of segmental duplication or common repeats (SD+CR). . . . 26
4.7 Performance of the offline set cover approach, in which reads are assigned to the cores by the greedy set cover algorithm. LCP performance is provided as well. File sizes are reported in megabytes. . . . 27
4.8 Performance of the online set cover approach, in which reads are assigned to the cores by the SCALCE based on set cover generated core substrings. LCP performance is provided as well. File sizes are reported in megabytes. . . . 27

List of Figures

3.1 Aho-Corasick trie preprocessing. Red edges indicate original failure edges, while green edges indicate preprocessed failure edges. Note that we need to perform only one jump by using green edges to reach the destination, contrary to the two jumps needed by red edges. 17

3.2 Original (left) and transformed (right) quality scores for two random reads that are chosen from NA18507 individual. The original scores show much variance, where the transformed quality scores are smoothened except for the peaks at local maxima, that help to improve the compression ratio. . . . 19

3.3 Frequency plot before (red) and after (blue) applying the greedy transformation method with different error thresholds. Increased lossiness threshold shows more smoothing (i.e. smaller alphabet size) in quality scores. 20

Chapter 1

Introduction

The recent introduction of high throughput sequencing (HTS) platforms enabled the fast and cheap sequencing of genomes of various species and individuals. This technology advancement enabled us to perform much broader genomic analysis in a shorter amount of time. On the other side, these platforms generate unprecedented amounts of data that introduce challenges for the computational infrastructure, such as data management, storage, and analysis.

Although the vast majority of HTS data is compressed through general purpose methods, in particular gzip and its variants, the need for improved performance has recently lead to the development of a number of techniques designed specifically for HTS data. Available compression techniques for HTS data either exploit the similarity between the reads and a reference genome, or the similarity between the reads themselves. Once such similarities are established, each read is encoded by the use of techniques derived from classical lossless compression algorithms such as Lempel-Ziv-77 [40] (which is the basis of gzip and all other zip formats) or Lempel-Ziv-78 [41].

Compression methods that exploit the similarity between individual reads and the reference genome use the reference genome as a "dictionary" and represent individual reads with a pointer to one mapping position in the reference genome, together with additional information about whether the read has some differences with the mapping loci. As a result, these methods [21, 24] require both the availability of a reference genome and the mapping of the reads to the reference genome. Unfortunately, genome mapping is a time-wise costly step, especially when compared to the actual execution of compression (i.e. *encoding* of the reads) itself. Furthermore, these methods necessitate the availability of a reference genome

both for compression and decompression. Finally, many large-scale sequencing projects such as the Genome 10K Project [19] focus on species without reference genomes.

Compression methods that exploit the similarity between the reads themselves simply concatenate the reads to obtain a single sequence. For example, [6] apply modification of Lempel-Ziv algorithm, [36, 12] use Huffman Coding while [22] and [11] employ Burrows Wheeler transformation [7]. In particular, the Lempel-Ziv methods (e.g gzip and derivatives) iteratively go over the concatenated sequence and encode a prefix of the uncompressed portion by a “pointer” to an identical substring in the compressed portion. This general methodology has three major benefits: (i) Lempel-Ziv based methods (e.g. gzip and derivatives) have been optimized through many years and are typically very fast; in fact the more “compressible” the input sequence is, the faster they work, both in compression and decompression; (ii) these methods do not need a reference genome; and (iii) since these techniques are almost universally available, there is no need to distribute a newly developed compression algorithm.

Interestingly, the availability of a reference genome can improve the compression rate achieved by standard Lempel-Ziv techniques. If the reads are first mapped to a reference genome and then reordered with respect to the genomic coordinates they map to before they are concatenated, they are not only compressed more due to increased locality, but also in less time. This, mapping first compressing later approach, combines some of the advantages of the two distinct sets of methods above: (a) it does not necessitate the availability of a reference genome during decompression (compression is typically applied once to a data set, but decompression can be applied many times), and (b) it only uses the re-ordering idea as a front end *booster* (for example, Burrows Wheeler transform – BWT – is a classical example for a compression booster. It rearranges input symbols to improve the compression achieved by Run Length Encoding and Arithmetic Coding. Further boosting for BWT is also possible: see [17, 16, 15]). Any well-known, well-distributed compression software can be applied to the re-ordered reads. Unfortunately, this strategy still suffers from the need for a reference genome during compression.

In this thesis we introduce a novel HTS genome (or transcriptome, exome, etc.) sequence compression approach that combines the advantages of the two types of algorithms above. It is based on re-organization of the reads so as to ”boost” the locality of reference. The re-organization is achieved by observing sufficiently long “core” substrings that are shared between the reads, and clustering such reads to be compressed together. This reorganization

acts as a very fast substitute for mapping based reordering (see above); in fact the first step of all standard seed and extend type mapping methods identify blocks of identity between the reads and the reference genome.

The core substrings of our boosting method are derived from the Locally Consistent Parsing (LCP) method devised by Sahinalp and colleagues [32, 10, 5]. LCP is a combinatorial pattern matching technique that aims to identify "building blocks" of strings. It has been devised for pattern matching, and provides faster solutions in comparison to the quadratic running time offered by the classical dynamic programming schemes. As a novel application, we introduce LCP to genome compression, where it aims to act as a front end (i.e. booster) for commonly available data compression programs. For each read, LCP simply identifies the longest core substring (there could be one or more cores in each read). 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. We compress reads in each bucket using Lempel-Ziv variants or any other related method without the need for a reference genome.

As can be seen, LCP mimics the mapping step of the mapping-based strategy described above in an intelligent manner: on any pair of reads with significant (suffix-prefix) overlaps, LCP identifies the same core substring and subsequently buckets the two reads together. For a given read, the recognition of the core strings and bucketing can be done in time linear with the read length. Note that the "dictionary" of core substrings is devised once for a given read length as a pre-processing step. Thus, the LCP-based booster we are proposing is very efficient. LCP provides mathematical guarantees that enable highly efficient and reliable bucketing that captures substring similarities.

In this study, we apply the LCP-based reordering scheme for (i) short reads of length 51 bp obtained from bacterial genomes and (ii) short reads of length 100 bp from one human genome. We achieve significant improvements in both compression rate and running time over alternative methods.

We also provide a theoretical insight into the general string reordering problem by reducing it to the set cover problem. In fact, we devise an alternative algorithm for the core substring finding, based on the well-known approximation algorithm for set cover [8]. We analyse the performance of LCP-based method in comparison to the alternative proposed method , and show that LCP version offers competitive or better results in practice, while requiring smaller computational resources.

Chapter 2

Preliminaries

This section provides a short introduction to the basic terminology and methods which will be employed later. In particular, the introduction to HTS technology, as well as the introduction to some compression techniques will be provided.

2.1 High-throughput sequencing

DNA sequencing is a process of precisely determining the order of nucleotides within a given DNA molecule. The ultimate goal of DNA sequencing is to precisely identify the content of the genome within various species and individuals. With the current sequencing technologies, only small chunks of DNA molecule (called *reads*) are extracted at a given time. In order to reconstruct the original DNA molecule from the given reads, methods such as *de novo* assembly are utilized. High-throughput sequencing is a technology where multiple reads can be extracted at once in a parallel manner. Usually, the magnitude of extracted reads in the same time is measured in thousands. There are different technologies which deal with high throughput sequencing, and in this work we will mainly focus on Illumina sequencing technologies. In aforementioned technologies, sequencing machine will provide a bunch of short reads of the same length. Currently, read lengths are usually spanned from 50 to 250 nucleotides (also called *basepairs*, abbreviated as bp). As these sequencing technologies are not yet error-prone, each basepair is assigned a quality score, which describes the accuracy level of the given basepair call. Because of the possible errors, each location in the DNA molecule is usually sequenced multiple times. The average number of reads covering the same location within a given DNA molecule is called *coverage*. Coverage

is variable which depends on the equipment and the settings used during the sequencing process. Obviously, higher coverage will result in higher number of sequenced reads. While it will provide a higher accuracy, it will also yield the increased storage requirements. Thus, for sequencing of the human individual (whose genome size is approximately 3 GB) with 40x coverage, we will need around 300 GB of storage space in order to store the reads and their corresponding qualities. Such storage requirements are simply impractical for most purposes, which naturally yields the question of reducing the size of HTS data.

2.2 LZ77 family of compression algorithms

LZ77 is a lossless data compression algorithm developed by A. Lempel And J. Ziv [40]. It is representative of the dictionary-based compression algorithm family.

Let us denote the input sequence as S. In LZ77 algorithm, dictionary is dynamic, and it is represented by a sliding window W of size n. For each input symbol S_i which is about to be encoded, the window W consists of n input symbols occurring before the symbol S_i; i.e. $W = S_{i-n}, \ldots, S_{i-1}$. The main idea of LZ77 is to find the largest number k such that a sequence of the symbols $S_i, \ldots, S_{i+k}$ is found in the dictionary W. In other words, there is an index j such that sequences $S_i, \ldots, S_{i+k}$ and $W_j, \ldots, W_{j+k}$ are equal. For each symbol, LZ77 will output the triple (k, j, c), where c is the symbol S_{i+k+1}, i.e. the symbol after the encoded subsequence.

The LZ77 algorithm is proven to be asymptomatically optimal algorithm for stationary ergodic sources. From the discussion above, it is clear that the performance of LZ77 is affected by the sliding window length n, as well as by the similarity of neighbouring subsequences in the input. Due to the speed concerns and performance of the triplet encoding, most practical implementations use a small dictionary of size 32 KB. Thus, possible reordering of the input sequence in a way that similar or equal subsequences are clustered together can drastically improve the performance of LZ77 algorithm.

2.3 A theoretical exposition to the LCP technique.

The simplest form of the LCP technique works only on reads that involve no tandemly repeated blocks (i.e. the reads can not include a substring of the form XX where X is a string of any length ≥ 1; note that a more general version of LCP that does not require

this restriction is described in [31, 32, 5], so that in practice LCP works on any string of any length). Under this restriction, given the alphabet $\{0, 1, 2, \ldots, k-1\}$, LCP partitions a given string S into non-overlapping blocks of size at least 2 and at most k such that two identical substrings R_1 and R_2 of S are partitioned identically, except for a constant number of symbols on the margins. LCP achieves this by simply marking all local maxima (i.e. symbols whose value is greater than its both neighbours) and all local minima which do not have a neighbour already marked as a local maxima – note that beginning of S and the ending of S are considered to be special symbols lexicographically smaller than any other symbol. LCP puts a block divider after each marked symbol and the implied blocks will be of desirable length and will satisfy the identical partitioning property mentioned above. Then, LCP extends each block residing between two neighbouring block dividers by one symbol to the right and one symbol to the left to obtain *core substrings* of S. Note that two neighbouring core substrings overlap by two symbols.

Example. Let

$$S = 21312032102\mathbf{0}21312032102;$$

in other words $S = X\mathbf{0}X$, where $X = 21312032102$. The string S satisfies the above condition; i.e. it contains no identically and tandemly repeated substrings. When the above simple version of LCP is applied to S, it will be partitioned as

|213|12|03|2102|**0**2|13|12|03|2102|.

Clearly, with the exception of the leftmost blocks, the two occurrences of X are partitioned identically. Now LCP identifies the core substrings as

2131, 3**12**0, 2**03**2, 3**2102**0, 2**02**1, 2**13**1, 3**12**0, 2**03**2, 3**2102**.

Observe that:

1. the two occurrences of string X are partitioned by LCP the same way except in the margins.

2. if a string is identified as a core substring in a particular location, it must be identified as a core substring elsewhere due to the fact that all symbols that lead LCP to identify that block as a core substring are included in the core substring.

3. as a result, all core substrings that entirely reside in one occurrence of X should be identical to those that reside in another occurrence of X

4. the number of cores that reside in any substring X is at most $1/2$ of its length and at least $1/k$ of its length

The above version of LCP can return core substrings with length as small as 4. Length 4 substring is clearly not specific enough for clustering an HTS read, so we have to ensure that the minimum core substring length c is a substantial fraction of the read length. LCP as described in [31, 32, 5] enables to partition S into non-overlapping blocks of size at least c and at most $2c-1$ for any user defined c. These blocks can be extended by a constant number of symbols to the right and to the left to obtain the core substrings of S.

We show here how to increase the minimum core substring length to some 2^i for any desired i on any string that includes no tandemly repeated substrings. Although this version of LCP is not the most general one (the upper bound on the core length will not be $2^{i+1}-1$, but rather 3^i and the restriction of tandemly repeated substrings is substantial), it will be sufficient in developing a practical version of LCP explained later.

Given a sufficiently long string (HTS read) S that includes no tandemly repeated substrings, LCP aims to identify all core substrings of length 2^i to 3^i by building an edge ordered 2-3 tree. An edge ordered 2-3 tree is one in which all leaves have the same distance from the root and each internal node has either 2 or 3 children - which are ordered. Each leaf of this 2-3 tree is labelled by one of the symbols of the read; in fact the ordered list of leaf labels of the 2-3 tree gives (at least a significantly long substring of) the read itself. Each internal node n of the 2-3 tree is also labelled, not by the DNA alphabet in use but rather by novel symbols, in a way to reflect the substring represented by the leaves of the subtree rooted at n: two internal nodes of the 2-3 tree have the same labels if and only if the substrings of their leaf level descendants represent are the same.

Given the leaves of the 2-3 tree to be constructed, the parents of the leaf nodes are obtained as follows (note that this procedure will be iteratively applied to any level of the 2-3 tree after all nodes and their labels are obtained at that level). For any symbol s of a leaf, LCP determines whether to put a block divider after it. To be able to do that, LCP uses the two right neighbours and $max(\log^*(d)-1;5)$[1] left neighbours of that symbol -

[1] $\log^*(d)$, usually read as "log star", is the iterated logarithm of d that is the number of times the logarithm function must be iteratively applied before the result is less than or equal to 1

where d is the number of unique node labels. A property of LCP is that the block dividers partition the sequence into blocks whose size is always 2 or 3. Now, the algorithm proceeds as follows:

1. for each symbol in the sequence, LCP computes its tag. The tag of a symbol is the index of the least significant bit that differs between the binary representations of the symbol and its left neighbour.

2. each symbol is replaced by a number obtained by the concatenation of two binary numbers: (i) the tag of the symbol and (ii) the value of the bit of the symbol whose index is tag. If the initial range of symbols are $0, \ldots, d$, they will be replaced by numbers selected from $0, \ldots, 2 \log d$.

LCP applies the same procedure for $\log^*(d) - 1$ times. In the first iteration the range of symbols would be reduced from $0, \ldots, d-1$ to $2 \log d - 1$, in the second iteration, to $0, \ldots, 2 \log(\log d) - 1\ + 2$, and in the last iteration to $0, \ldots, 5$.

Following this final iteration, we will end up with a string S' composed of symbols from the alphabet of size 6. Furthermore S', as per S, will have no tandemly repeated substrings. Thus we can apply the simple version of LCP described above to partition S' into blocks of size 2 to 6. We further partition any block of size 4 to two blocks of size 2 each, any block of size 5 to blocks of size 2 and 3 and any block of size 6 to two blocks of size 3 each. As mentioned above, once the blocks of length 2 or 3 are obtained, a parent node is obtained for each block and it is given a label such that two parent nodes have the same labels if and only if they represent the same block.

Notice that LCP decides to put a block divider after a symbol or not, by looking at only $\log^*(d) - 1$ left neighbors of that symbol. This implies that two identical substrings would be partitioned identically except in the very left and right margins of the sequences as per Lemma 1 below.

Lemma 1 (Consistent Parsing Lemma)**.** *Let $S_{i,i+k}$ and $S_{j,j+k}$ be identical substrings of S. We call substring $S_{i+\log^* d, i+k-3}$, the interior of $S_{i,i+k}$. Similarly, $S_{j+\log^* d, j+k-3}$ is the interior of $S_{j,j+k}$. Then, LCP will derive the same blocks from the interiors of $S_{i,i+k}$ and $S_{j,j+k}$.*

Proof. Consider the symbol $S_{i+\log^* d-1}$. LCP needs $\log^* d - 2$ iterations for deciding to put a block divider after this symbol. The tags of this symbol are a function of $S_{i+\log^* d-2, i+\log^* d}$

in the first iteration, $S_{i+\log^* d-3,i+\log^* d}$ in the second iteration and $S_{i+1,i+\log^* d}$ in the last iteration. However $S_{i,i+\log^* d}$ is identical with $S_{j,j+\log^* d}$. Hence, LCP should work identically for symbols $S_{i+\log^* d}$ and $S_{j+\log^* d}$, and those symbols that are to their right. □

It is possible to improve the running time of LCP to $O(1)$ per symbol: after the first iteration of LCP, one could emulate its remaining iterations in one step by using a table of size $O(2\log^{\log^* d} d)$ which keeps the results of applications of LCP to all possible sequences of size $\log^* d$, selected from the range $0, \ldots, 2\log d - 1$.

As can be seen, the above version of LCP guarantees that each internal node of level k of the 2-3 tree represents a substring of length in the range $\left[2^k, 3^k\right]$. Furthermore each string of length ℓ is guaranteed to have a 2-3 tree with at least one node at level $\log_3 \ell - O(\log^* d)$ - which is guaranteed to cover at least $2^{\log_3 \ell - O(\log^* d)}$ symbols of the string. The substrings represented by the nodes of this level of the 2-3 tree are thus called the core substrings of the input.

Batu et al. introduced a much improved variant of LCP to compute core substrings of an input string of length $4c-8$, where the core substrings have lengths in the range $[c, 2c-1]$. This variant also allows the presence of tandemly repeated substrings in the input string. Unfortunately because it is highly complex and only works for sufficiently large values of c, it is only of theoretical interest.

In the context of compressing HTS reads, if c is picked to be a significantly long fraction of the read size, LCP applied on the HTS reads will guarantee that each read will include at least one and at most three of these core substrings.

2.4 Arithmetic encoding

Arithmetic encoding [30, 38, 33] is a lossless entropy encoding scheme, whose main idea is to represent the whole input sequence as a fraction n, where $0 \leq n \leq 1$. For each input symbol, arithmetic encoding algorithm needs to know a probability assigned to that symbol in order to successfully create the output fraction n. In order for data to be decoded, both encoder and decoder have to use same probability distributions for the input symbols. The technique of assigning probabilities to the symbols is usually referred as *data modelling*.

Arithmetic encoder's performance directly depends on the underlying data model. In fact, it is well known result that arithmetic encoder can achieve optimal compression performance (i.e. the compressed size is close to the entropy of the input data) with appropriate

model. Unfortunately, modelling problem is not uniform, since different data sources usually have different optimal models. The most commonly used models are simple order-n models, where for calculating the probability of the input symbol p_i, we also take into account the last n symbols which occurred before p_i, namely $p_{i-1}, \ldots, p_{i-n}$. These symbols are also known as a *context* of the symbol p_i. These models can be either stationary or adaptive. For stationary models, probabilities of input symbols are calculated and known in advance of the encoding process, while in the case of adaptive models, probabilities are constantly being updated as the new symbols arrive to the coder. Obviously, for the streaming data applications, adaptive models are the only viable choice.

2.5 FASTQ files

Currently, the industry standard for storing HTS data is FASTQ format [9], which is based on a previous FASTA format [29]. It is a plain text based format, in which each HTS read is represented with the four plain ASCII lines, where:

1. first line represents the unique read name, and it always starts with the symbol `@`
2. second line represents the nucleotide sequence (usually with letters A, C, T, G, N, where N stands for an unidentified nucleotide)
3. third line represents the comment, and it always starts with the symbol `+` or `-`
4. fourth line represents the quality score, which is a sequence of printable ASCII characters

HTS data is usually stored in a FASTQ libraries, where each FASTQ file usually contains a fixed number of reads. Some HTS sequencers produce paired-end data, where each FASTQ file is associated with another FASTQ file which contains the same read names, and have the same number and order of the reads (i.e. only the sequences and qualities differ).

According to the FASTQ specification [9], order of the reads does not matter at all. Also, currently available sequencers do not produce reads in any usable order (in fact, the reads are produced in a random fashion). This implies that reordering of the FASTQ files is completely harmless operation. Only thing which has to be taken care of during the reordering is that, in the case of the paired-end FASTQ files, both ends should be reordered in the same way (as stated, second file has to be ordered exactly as the first file).

As for the compression purposes, comment line is usually redundant and can be safely removed. In most practical uses, even the names can be omitted, as long as we can distinguish the different reads uniquely. In the case of paired-end libraries, if we want to drop the names, we need to ensure that both ends will still have the same read names after decompression (as the only criteria of knowing which reads are paired is the read name equality).

Chapter 3

Methods

3.1 Problem definition

The main goal of our method is to improve the compression performance of Lempel-Ziv family of algorithms through reordering of the reads. The purpose of reordering is to cluster highly related reads (in fact those reads that ideally come from the same region and have large overlaps) so as to boost gzip and other Lempel-Ziv-77 based compression methods. If one concatenates reads from a donor genome in an arbitrary order, highly similar reads will be scattered over the resulting string. Because Lempel-Ziv-77 based techniques compress the input string iteratively, from left to right, replacing the longest possible prefix of the uncompressed portion of the input string with a pointer to its earlier (already compressed) occurrence, as the distance between the two occurrences of this substring to be compressed increases, the binary representation of the pointer also increases. As a result gzip and other variants only search for occurrences of strings within a relatively small window, as explained before. Thus reordering reads for purpose of bringing together those reads with large (suffix–prefix) overlaps is highly beneficial to gzip and other similar compression methods. For this purpose, it is possible to reorder the reads by sorting them based on their mapping loci on the reference genome. Alternatively it may be possible to find similarities between the reads through pairwise comparisons [39]. However each one of these approaches are time-wise costly.

Thus, as previously mentioned, we opt for the reordering based on shared core substrings. The key problem is to find a set of core substrings S containing the minimal number of strings of limited size (at least S_L and at most S_H), such that each read contains at least one core

substring from set S. This represents the maximum parsimony principle, and the intuition behind this choice is that each new cluster of reads with the same core substring will yield a "dictionary reset" of LZ method, which will result in lower compression rate. Thus selecting fewer core substrings will yield fewer dictionary resets. We can formally define this problem as follows:

Problem 1. Let Σ be an alphabet, R set of input strings on Σ, and let $S_L \leq S_H$ be two integers where $S_H \leq \min_{r \in R} |r|$. Find a subset S of the set of all substrings x of size $S_L \leq |x| \leq S_H$ on the alphabet Σ such that:

- for any $r \in R$, r contains a substring from the S
- there does not exist a smaller set S' which satisfies the conditions above

The condition $S_L \leq |x| \leq S_H$ from the problem 1 is redundant, since we can always select strings of size S_L which will satisfy the conditions above. In order to allow larger core sizes, we can assign each core substring s_i a weight $w(s_i)$. In that way we introduce a "weighted" version of the problem 1, where the objective is to minimize the total sum of weights of elements in the resulting set S. Note that we can also filter various highly repeated core substrings (like poly-As) by using appropriate weights.

3.2 Set cover approach for identifying the core substrings

We can reduce the problem 1 to an instance of the set cover problem. Let the universe be a set of all reads $U = \{r_1, \ldots, r_N\}$. Let the set S represent a set of all core subsets, where by core subset of string s we define a set of all reads which contain s as a substring. It becomes clear that the cores obtained from the minimal subset of S whose union "covers" U will satisfy the conditions as stated in the definition of problem 1.

Set cover problem is known to be NP-Hard [28], but fortunately this problem can be solved by using a simple greedy algorithm from [8], which has the best known approximation of factor $O(\log |S|)$. Note that this approach does not only detect the core substrings based on the optimal set cover, but it also immediately provides an optimal clustering scheme, as it assigns each read to its representative core.

Unfortunately, the construction of the sets S and U is memory-expensive operation, since the number of input reads in HTS datasets is measured in tens of millions. Thus, in

this thesis we utilized a different approach, based on Locally Consistent Parsing, which we used for finding of the core substrings.

3.3 A practical implementation of LCP for reordering reads.

As we stated previously, the general version of LCP is too complex to be of practical interest. Thus, we have developed a practical variant of LCP described below to obtain core substrings of each HTS read with minimum length 8 and maximum length 20. Interestingly we observed that in practice more than 99% of all HTS reads of length 50 or more include at least one core of length 14 or less. As a result, we are interested in identifying only those core substrings of lengths in the range $[8, 14]$. Still there could be multiple such core substrings in each HTS read; we will pick the longest one as the *representative core substring* of the read (if there are more than one such substring, we may break the tie in any consistent way). Then, this read will be clustered with other reads that have the same representative core substring.

As previously stated, our goal here is to obtain a few core substrings for each read so that two highly overlapping reads will have common core substrings. The reads will be reordered based on their common core substrings which satisfy the following properties:

1. each HTS read includes at least one core substring.
2. each HTS read includes at most a small number of core substrings.

This would be achieved if any sufficiently "long" prefix of a core substring can not be a suffix of another core substring (this assures that two subsequent core substrings can not be too close to each other).

We first extend the simple variant of LCP described above so as to handle strings from the alphabet $\Sigma = \{0, 1, 2, 3\}$ (0=A, 1=C, 2=G, 3=T) that *can* include tandemly repeated blocks. In this variant we define a core substrings as any 4-mer that satisfies one of the following rules:

- (Local Maxima) $xyzw$ where $x < y$ and $z < y$
- (Low Periodicity) $xyyz$ where $x \neq y$ and $z \neq y$
- (Lack of Maxima) $xyzw$ where $x \neq y$ and $y < z < w$
- (Periodic Substrings) $yyyx$ where $x \neq y$

We computed all possible 4-mers (there are 256 of them) from the 4 letter alphabet Σ and obtained 116 core substrings that satisfy the rules above. The reader can observe that the minimum distance between any two neighbouring cores will be 2 and the maximum possible distance will be 6.[1]. This ensures that any read of length at least 9 includes one such core substring.

In order to capture longer regions of similarity between reads, we need to increase the lengths of core substrings. For that purpose we first identify the so called marker symbols in the read processed as follows. Let x, y, z, w, x, $v \in \Sigma$, then:

- y is a "marker" for xyz, when $x < y$ and $z < y$
- y is a "marker" for $xyyz$, when $x < y$ and $z < y$
- y is a "marker" for $xyyyz$, when $x \neq y$ and $z \neq y$
- yy is a "marker" for $xyyyyz$, when $x \neq y$ and $z \neq y$
- y is a "marker" for $xwyzv$, when $y < w \leq x$ and $y < z \leq v$

Now on a given read, we first identify all marker symbols. We apply LCP to the sequence obtained by concatenating these marker symbols to obtain the core substrings of the marker symbols. We then map these core substrings of the marker symbols to the original symbols to obtain the core substrings of the original read.

Example Given read $R = 0230000300$, we identify its marker symbols as follows: 3 is the marker for 230, 00 is the marker for 300003, and 3 is the marker for 030 as per the marker identification rules above. The sequence obtained by concatenating these markers is 3003, which is itself (4-mer) core substring according to the LCP description above. The projection of this core substring on R is 23000030, which is thus identified as a core substring (actually the only core substring) of the read.

For the 4 letter alphabet Σ, we computed all (approximately 5 million) possible core substrings of length $\{8, \ldots, 14\}$ according to the above rules.[2] These rules assure that the minimum distance between two subsequent core substrings is 4 and thus the maximum

[1] Note that this implementation of LCP is not aimed to satisfy any theoretical guarantee; rather, it is developed to work well in practice.

[2] This is about 1% of all substrings in this length range.

number of core substrings per read is at most 11 per each HTS read of length 50. Furthermore we observed that more than 99.5% of all reads have at least one core substring (the other reads have all cores of length 15 to 20). Although this guarantee is weaker than the theoretical guarantee provided by the most general version of LCP, it serves our purposes.

3.4 A data structure for identifying core substrings of reads.

We build a trie data structure representing each possible core substring by a path in the trie to efficiently place reads into the corresponding bucket. First we find all core substrings of each read and place the read in the bucket associated with the largest found core substring. If there are multiple largest core substrings, we choose the one whose bucket contains the maximum number of reads (if there are two or more such buckets, we pick one arbitrarily in a consistent way).

If one simply uses the trie data structure, finding all core substrings within a read would require $O(cr)$ time where r is the read length, and c is the length of all core substrings in that read. To improve the running time we build an automaton implementing the Aho-Corasick dictionary matching algorithm [2]. This improves the running time to $O(r + k)$, where k is the number of core substring occurrences in each read. Since the size of the alphabet Σ is very small (4 symbols), and the number of core substrings is fixed, we can further improve the running time by pre-processing the automaton such that, for a given state of the automaton, we calculate the associated bucket in $O(1)$ time, reducing the total search time to $O(r)$. This was done by analysing the original Aho-Corasick trie and redirecting the backward (also called failure) edges to the ultimate source edges (and thus reducing the number of steps for failure look-up), as shown in figure 3.4.

In order to improve the compression of the reads even more, we apply cyclic shifting based on the core position in each read, and then sort each bucket alphabetically, in order to boost the similarity between the neighbouring sequences even more.

3.5 Compressing the quality scores.

Note that the HTS platforms generate additional information for each read that is not confined to the 4 letter alphabet Σ. Each read is associated with a secondary string that

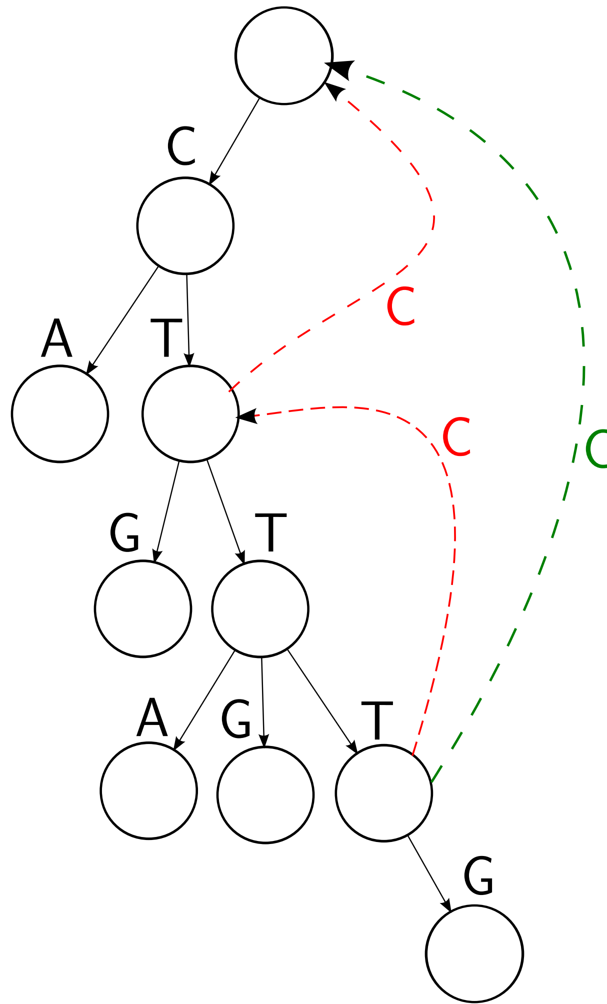

Figure 3.1: Aho-Corasick trie preprocessing. Red edges indicate original failure edges, while green edges indicate preprocessed failure edges. Note that we need to perform only one jump by using green edges to reach the destination, contrary to the two jumps needed by red edges.

contains the base calling *phred* [14] quality score. Quality score of a base defines the probability that the base call is incorrect, and it is formulated as $Q = -10 \times \log_{10}(P(error))$ [14] in Illumina platforms. The size of the alphabet for the quality scores is typically $|\Sigma| = 40$ for the Illumina platforms, yielding much lower compression rate for quality scores compared to the actual reads compression rate.

As mentioned in previous studies [37], lossy compression can improve the quality scores compression rate. We provide an optional controlled lossy transformation approach based on the following observation. In most cases, for any basepair b, the quality scores of its "neighbouring" basepairs would be either the same or within some small range of b's score (see Figure 3.2). Based on this observation, we provide a lossy transformation scheme to reduce the alphabet size. We calculate the frequency table for the alphabet of quality scores from a reasonable subset of the qualities (100,000 quality scores, although this number can be further tuned by the user). Then we use a simple greedy algorithm to find all the local maxima within this table. Afterwards, we reduce the variability among the quality scores in the vicinity of local maxima up to some error threshold e (i.e. such that each quality value within $e\%$ of the local maxima is replaced with a quality assigned to that local maxima). At the end, quality scores are encoded with the arithmetic encoding, which uses simple order-2 model.

Note that, by default, lossy option is not used, and it is completely left for the user to choose the lossy compression level. The reason behind this is that some downstream analysis applications can misbehave if the quality scores are changed.

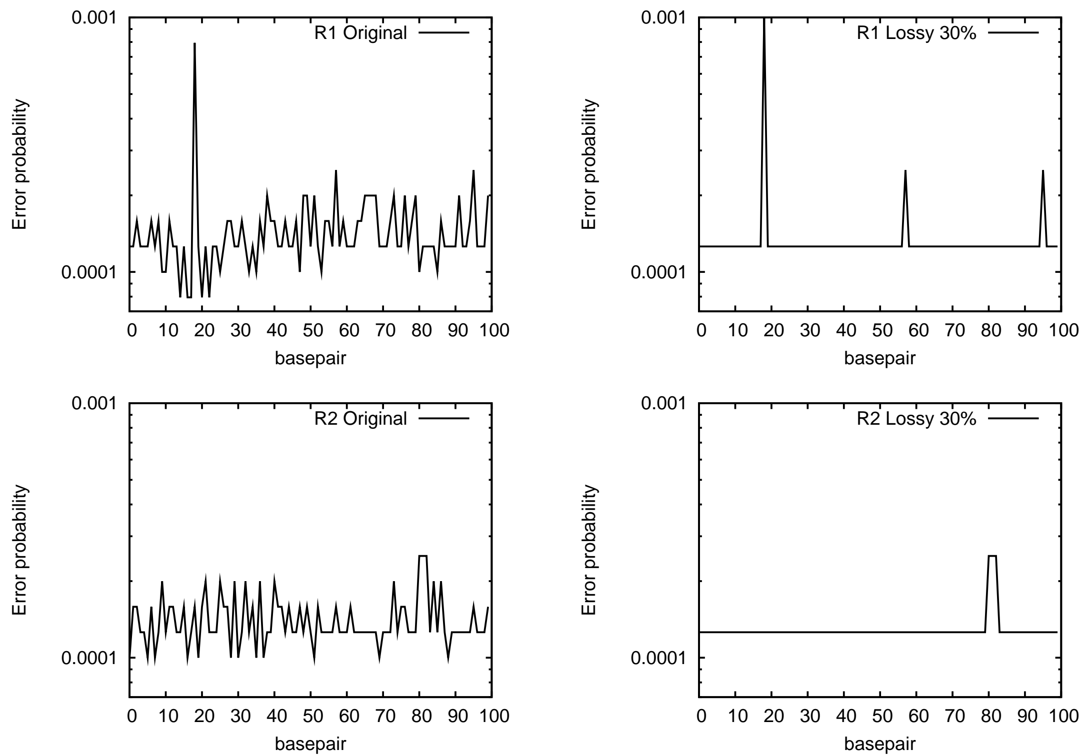

Figure 3.2: Original (left) and transformed (right) quality scores for two random reads that are chosen from NA18507 individual. The original scores show much variance, where the transformed quality scores are smoothened except for the peaks at local maxima, that help to improve the compression ratio.

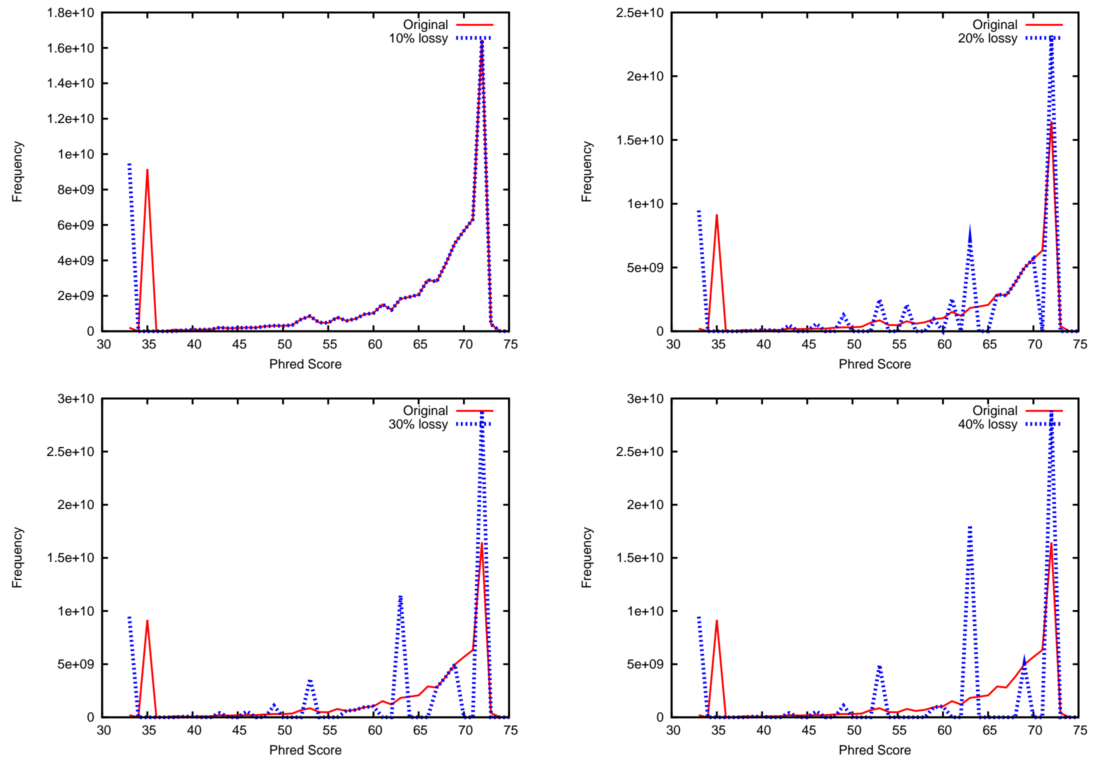

Figure 3.3: Frequency plot before (red) and after (blue) applying the greedy transformation method with different error thresholds. Increased lossiness threshold shows more smoothing (i.e. smaller alphabet size) in quality scores.

Chapter 4

Results

All of the mentioned ideas are gathered together in the compression tool named SCALCE, which was implemented in C++. We evaluated the performance of the SCALCE algorithm for boosting gzip on a single core 2.4GHz Intel Xeon X5690 PC (with network storage and 6GB of memory).

We used four different data sets in our tests:

1. P. aeruginosa RNA-Seq library (51 bp, single lane)
2. P. aeruginosa genomic sequence library (51 bp, single lane)
3. whole genome shotgun sequencing (WGS) library generated from the genome of the HapMap individual NA18507 (100 bp reads at 40x genome coverage), and
4. a single lane from the same human WGS data set corresponding to approximately 1.22x genome coverage (SRA ID: SRR034940).

We removed any comments from name section (any string that appears after the first space). Also the third row should contain a single character (+/-) separator character.

The reads from each data set were reordered and compressed through SCALCE and three separate files were obtained for:

1. the reads themselves,
2. the quality scores and
3. the read names.

Table 4.1: Input data statistics and compression rates achieved by gzip only and SCALCE + gzip on reads from the P. aeruginosa RNA-Seq library. File sizes are reported in megabytes.

Original size	**4,327**
Number of reads	89 million
gzip size	**1,071**
gzip rate	4.04
gzip time	13m 18s
SCALCE+gzip size	**256**
SCALCE + gzip rate	16.92
SCALCE boosting factor	4.19x
gzip time after reordering	53s
SCALCE+gzip time	6m 21s

Each of the files maintains the same order of the reads.

Note that LCP reordering is useful primarily for compressing the reads themselves through gzip. The quality scores were compressed via the scheme described above. Finally the read names were compressed through gzip as well.

The compression rate and run time achieved by gzip software alone, only on the reads from the P. aeruginosa RNA-Seq library (data set 1) is compared against those achieved by SCALCE followed by gzip in Table 4.1.

The compression rates achieved by the gzip software alone in comparison to gzip following SCALCE on the combination of reads, quality scores and read names are presented in Table 4.2.

The run times for the two schemes (again on reads, quality scores and read names all together) are presented in Table 4.3.

When SCALCE is used with arithmetical coding of order 2 with lossless qualities, it boosts the compression rate of gzip between $1.42-2.13$-fold (when applied to reads, quality scores and read names), significantly reducing the storage requirements for HTS data. When arithmetical coding of order 2 is used with 30% loss – without reducing the mapping accuracy – improvements in compression rate are between $1.86-3.34$. In fact, the boosting factor can go up to 4.19 when compressing the reads only. Moreover, the speed of the gzip compression step can be improved by a factor of 15.06. Interestingly the total run time for SCALCE + gzip is less than the run time of gzip by a factor of 2.09. Furthermore, users can tune the memory available to SCALCE through a parameter to improve the run time when a large main memory is available. In our tests, we limited the memory usage to 6GB.

Table 4.2: Input data statistics and compression rates achieved by gzip only and SCALCE + gzip + AC on complete FASTQ files. File sizes are reported in megabytes.

	P. aeruginosa RNAseq	P. aeruginosa Genomic	NA18507 WGS	NA18507 Single Lane
Original size	**10,076**	**9,163**	**300,337**	**7,708**
Number of reads	89 million	81 million	1.4 billion	36 million
gzip performance				
Size	**3,183**	**3,211**	**113,132**	**3,058**
Rate	3.17	2.85	2.65	2.52
SCALCE performance with lossless qualities				
Size	**1,496**	**1,655**	**76,890**	**2,146**
Rate	6.74	5.54	3.91	3.59
Boosting factor	2.13x	1.94x	1.47x	1.42x
SCALCE performance with 30% lossy setting applied				
Size	**953**	**1,126**	**58,031**	**1,639**
Rate	10.58	8.14	5.18	4.70
Boosting factor	3.34x	2.85x	1.95x	1.86x

Our tests showed that SCALCE (when considering only reads) outperforms BEETL [11] combined with bzip2 by a factor between $1.09 - 2.07$, where running time is improved by a factor between $3.60 - 5.17$ (see Table 4.5). SCALCE (on full FASTQ files) also outperforms DSRC [12] compression ratio on complete FASTQ files by a factor between $1.09 - 1.18$ (see Table 4.4).

Note that our goal here is to devise a very fast boosting method, which in combination with gzip gives compression rates much better than gzip alone. It is possible to get better compression rates through mapping based strategies but these methods are several orders of magnitude slower than SCALCE+gzip.

Thus, as a final benchmark, we compared the performance of SCALCE with mapping based reordering before gzip compression. We first mapped one lane of sequence data from the genome of NA18507 (same as above) to human reference genome (GRCh37) using BWA [25], and sorted the mapped reads using samtools [26], and reconverted the map-sorted BAM file back to FASTQ using Picard (`http://picard.sourceforge.net`), resulting in raw FASTQ files of size $7,964$ MB. We then used the gzip tool to compress the map-sorted file to $3,091.5$ MB, achieving 2.57-fold compression rate. The preprocessing step for mapping and sorting required 18.2 CPU hours, and FASTQ conversion required 30 minutes, while compression was completed in 28 minutes. Moreover, the mapping based sorting did not

Table 4.3: Run time for running gzip alone and SCALCE + gzip + AC on complete FASTQ files.

	P. aeruginosa RNAseq	P. aeruginosa Genomic	NA18507 WGS	NA18507 Single Lane
gzip	**20m**	**20m**	**10h 52m**	**18m**
SCALCE+gzip+AC, single thread				
SCALCE reordering	7m	6m	3h	5m
gzip+AC	6m	5m	3h 1m	5m
Total	**13m**	**11m**	**6h 1m**	**10m**
SCALCE+gzip+AC, 3 threads				
Total	**9m**	**9m**	**4h 28m**	**7m 32s**

Table 4.4: Comparison of single-threaded SCALCE with DSRC. DSRC was tested using the `-l` option except on the WGS sample, where it crashed. Instead we had to use a faster but less powerful setting for this data set.

	P. aeruginosa RNAseq	P. aeruginosa Genomic	NA18507 WGS	NA18507 Single Lane
DSRC size	**1,767**	**1,846**	**94,707**	**2,341**
DSRC time	12m	6m	3h 16m	4m
SCALCE size	**1,496**	**1,655**	**76,890**	**2,146**
SCALCE time	13m	11m	6h 1m	10m

improve the compression run time even if we do not factor in the preprocessing. In contrast, SCALCE+gzip generated a much smaller file in less amount of time, with no mapping based preprocessing. We then repeated this experiment on the entire WGS data set (NA18507). The mapping based preprocessing took 700 CPU hours for BWA+samtools, and 10 CPU hours for Picard; gzip step was completed in 11 CPU hours, resulting in a compression rate of 4.93x. On the other hand, gzip needed only 6.5 CPU hours to compress the same data set (1.69x faster) after the preprocessing by SCALCE which took 8 CPU hours, and achieved a better compression rate (6-fold, Tables 4.2 and 4.3).

The run time of mapping based preprocessing step can be improved slightly through the use of BAM-file-based compressors such as CRAM tools [21], but this would reduce the time only by 10 CPU hours for the Picard step. Thus, in total, SCALCE+gzip is about 45 times faster than any potential mapping based scheme (including CRAM tools) on this data set.

Table 4.5: Comparison of single-threaded SCALCE with BEETL. Here, the data sets contained only reads from the FASTQ file, as BEETL supports only FASTA file format.

	P. aeruginosa RNAseq	P. aeruginosa Genomic	NA18507 Single Lane
BEETL size	**197**	**257**	**448**
BEETL time	29m	31m	51m
SCALCE size	**95**	**137**	**412**
SCALCE time	8m	6m	10m

4.1 Lossy compression effects

We tested the effects of the lossy compression schemes for the quality scores, employed by SCALCE as well as CRAM tools, to single nucleotide polymorphism (SNP) discovery. For that, we first mapped the NA18507 WGS data set with the original quality values to the human reference genome (GRCh37) using the BWA aligner [25], and called SNPs using the GATK software [13]. We repeated the same exercise with the reads after 30% lossy transformation of the base pair qualities with SCALCE. Note that the parameters for BWA and GATK we used in these experiments were exactly the same. We observed almost perfect correspondence between two experiments. In fact, $> 99.95\%$ of the discovered SNPs were the same (Table 4.6); not surprisingly most of the difference was due to SNPs in mapping to common repeats or segmental duplications. We then compared the differences of both SNP callsets with dbSNP Release 132 [35] in Table 4.6.

In addition, we carried out the same experiment with compressing/decompressing of the alignments with CRAM tools. As shown in Table 4.6, quality transformation of the CRAM tools introduced about 2.5% errors in SNP calling (97.5% accuracy) with respect to the calls made for the original data (set as the gold standard).

One interesting observation is that 70.7% of the new calls after SCALCE processing matched to entries in dbSNP where this ratio was only 62.75% for the new calls after CRAM tools quality transformation. Moreover, 57.95% of the SNPs that SCALCE “lost” are found in dbSNP, and CRAM tools processing caused removal of 18.4 times more potentially real SNPs than SCALCE.

Table 4.6: Number of SNPs found in the NA18507 genome using original qualities and transformed qualities with 30% noise reduction. Also reported are the number and percentage of novel SNPs in regions of segmental duplication or common repeats (SD+CR).

	# SNP Count	dbSNP v132	Novel	
			Total	in SD+CR
Original Qualities	4,296,152	4,092,923 (95.26%)	203,229	192,114 (94.53%)
Qualities using SCALCE	4,303,140	4,098,875(95.25%)	204,265	192,976 (94.47%)
Lost	7,931	4,596 (57.95%)	3,335	2,963 (88.84%)
New	14,919	10,548 (70.70%)	4,371	3,825 (87.51%)
Qualities using CRAM tools	4,202,298	4,013,401 (95.50%)	188,897	179,875 (95.22%)
Lost	101,957	84,607 (82.98%)	17,350	15,036 (86.66%)
New	8,103	5,085 (62.75%)	3,018	2,797 (92.67%)

4.2 Set cover approach

We also conduct comparison between the LCP-based method and the set cover technique for core substring finding. We evaluate a simplified version of the problem 1, in which $S_L = S_H$ and where $w(s_i) = 1$. This experiment is conducted on a single lane of NA18507 data (1.44x coverage), which is the only dataset we were able to utilize for this test, due to the high memory requirements. Note that we focus here only on HTS reads (not full FASTQ files).

In the first test, we evaluate the performance of the offline read assignment, in which reads are assigned to the respective cores by the greedy set cover algorithm, compared to the LCP-based approach. The results are shown in the table 4.7. Reordered reads are encoded with 2/8 encoding (i.e. each DNA symbol is encoded with 2 bits) and afterwards compressed with gzip.

In the second test, we provide the cores generated from the set cover experiment to SCALCE. In this case, read-to-core assignment is done by the SCALCE heuristics (in which, in addition to the 2/8 encoding, cyclic shifts and bucket sorting is applied as well). We will denote this as the online approach. Additionally, we use NA18507 generated cores with SCALCE for compressing the Illumina HiSeq sequenced human individual (4 lanes with 56x coverage, totalling 727,782,180 101bp reads), in order to see the performance of the generated human cores on a high-coverage human dataset. The results are shown in the table 4.8.

As we can see, LCP provides almost the same performance as the set cover method in

Table 4.7: Performance of the offline set cover approach, in which reads are assigned to the cores by the greedy set cover algorithm. LCP performance is provided as well. File sizes are reported in megabytes.

	gzip time	Size
LCP	26s	362
Set cover		
Core size 10	31s	374
Core size 11	31s	368
Core size 12	29s	364
Core size 13	28s	362
Core size 14	27s	360
Core size 15	26s	360

Table 4.8: Performance of the online set cover approach, in which reads are assigned to the cores by the SCALCE based on set cover generated core substrings. LCP performance is provided as well. File sizes are reported in megabytes.

	NA18507	HiSeq
LCP	347	17,201
Set cover		
Core size 10	353	19,998
Core size 11	350	19,906
Core size 12	339	19,587
Core size 13	340	18,962
Core size 14	341	18,829
Core size 15	343	19,224

the case of compressing single NA18507 lane. LCP uses much less memory and time, and its cores need to be generated only once, while set cover approach requires a new set of cores for every new genome (or set of input files).

Surprisingly, LCP cores completely outperform the set cover cores on the high-coverage HiSeq data. The possible reasons for this include the inability of our formulation of the problem 1 to cope with the repeated regions in the genome (which, in the case of human genome, appear quite often) and the simplicity of the weight function used for this experiment (for example, not allowing the variable core length). Also note that LCP is designed to work on *any* string on the given alphabet, while set cover cores are explicitly generated for the given reference genome, which might pose as a problem in a case when sequenced data contains a lot of sequencing errors or multiple genomic aberrations.

Chapter 5

Conclusion

The rate of increase in the amount of data produced by the HTS technologies is now faster than the Moore's Law [3]. This causes problems related to both data storage and transfer of data over a network. Traditional compression tools such as gzip and bzip2 are not optimized for efficiently reducing the HTS data files to manageable sizes in a short amount of time. To address this issue, several compression techniques have been developed with different strengths and limitations. For example, pairwise comparison of sequences can be used to increase similarity within "chunks" of data, thus increasing compression ratio [39], but this approach is also very time consuming. Alternatively, reference-based methods can be used, such as SlimGene [24] and CRAM tools [21]. Although these algorithms achieve very high compression rates, they have three major shortcomings. First, they require pre-mapped (and sorted) reads along with a reference genome, and this mapping stage can take very long time depending on the size of the reference genome. Second, speed and compression ratio are highly dependent on the mapping ratio since the unmapped reads are handled in a more costly manner (or completely discarded), which reduces the efficiency for genomes with high novel sequence insertions and organisms with incomplete reference genomes. Finally, the requirement of a reference sequence makes them unusable for *de novo* sequencing projects of the genomes of organisms where no such reference is available, for example, the Genome 10K Project [19].

The SCALCE algorithm provides a new and efficient way of reordering reads generated by the HTS platform to improve not only compression rate but also compression run time. Although it is not explored here, SCALCE can also be built into specialized alignment algorithms to improve mapping speed. We note that the names associated with each read do

not have any specific information and they can be discarded during compression. The only consideration here is that during decompression, new read names will need to be generated. These names need to be unique identifiers within a sequencing experiment, and the paired-end information must be easy to track. In fact, the Sequence Read Archive (SRA) developed by the International Nucleotide Sequence Database Collaboration adopts this approach to minimize the stored metadata, together with a lossy transformation of the base pair quality values similar to our approach [23]. However, in this paper we demonstrated that lossy compression of quality affects the analysis result, and although the difference is very small for SCALCE, this is an optional parameter in our implementation, and we leave the decision to the user.

Additional improvements in compression efficiency and speed may help ameliorate the data storage and management problems associated with high throughput sequencing [34].

5.1 Future work

SCALCE is mainly designed to tackle the compression rates of nucleotide sequences. Boosting the compression of quality scores is still an open problem and further investigation might yield a better performance. Compression of read names can also be improved by further analysis of the read name tokens, since in practice all read names in the same dataset share a common pattern. As the advancements of HTS technology focus mainly on the read size increase, we will need to devise novel methods for finding a longer core susbtrings, as well as methods for improved bucketing of the large reads.

The base problem 1 is also subject to further investigation. Designing the efficient set cover driven approach which utilises acceptable computational resources is still challenging. Also note that finding the optimal weight function for the problem 1 is an open problem, and that the role of repeats in the genome and their effect on the core generation needs to be further understood, as it might yield a deeper understanding why LCP offers better performance in some cases compared to the set cover approach.

Bibliography

[1] 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature 467*, 7319 (Oct 2010), 1061–1073.

[2] Aho, A. V., and Corasick, M. J. Efficient string matching: an aid to bibliographic search. *Commun. ACM 18*, 6 (1975), 333–340.

[3] Alkan, C., Coe, B. P., and Eichler, E. E. Genome structural variation discovery and genotyping. *Nat. Rev. Genet. 12*, 5 (May 2011), 363–376.

[4] Alkan, C., Kidd, J. M., Marques-Bonet, T., Aksay, G., Antonacci, F., Hormozdiari, F., Kitzman, J. O., Baker, C., Malig, M., Mutlu, O., Sahinalp, S. C., Gibbs, R. A., and Eichler, E. E. Personalized copy number and segmental duplication maps using next-generation sequencing. *Nature Genetics 41*, 10 (2009), 1061–1067.

[5] Batu, T., Ergün, F., and Sahinalp, S. C. Oblivious string embeddings and edit distance approximations. In *SODA 2006* (2006), pp. 792–801.

[6] Bhola, V., Bopardikar, A. S., Narayanan, R., Lee, K., and Ahn, T. No-reference compression of genomic data stored in fastq format. In *BIBM* (2011), pp. 147–150.

[7] Burrows, M., Wheeler, D. J., Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. Tech. rep., DEC Labs, 1994.

[8] Chvatal, V. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research 4*, 3 (1979), pp. 233–235.

[9] Cock, P. J. A., Fields, C. J., Goto, N., Heuer, M. L., and Rice, P. M. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic Acids Research 38*, 6 (2010), 1767–1771.

[10] Cormode, G., Paterson, M., Sahinalp, S. C., and Vishkin, U. Communication complexity of document exchange. In *SODA 2000* (2000), pp. 197–206.

[11] Cox, A. J., Bauer, M. J., Jakobi, T., and Rosone, G. Large-scale compression of genomic sequence databases with the burrows-wheeler transform. *Bioinformatics* (2012).

[12] Deorowicz, S., and Grabowski, S. Compression of dna sequence reads in fastq format. *Bioinformatics 27*, 6 (2011), 860–862.

[13] DePristo, M. A., Banks, E., Poplin, R., Garimella, K. V., Maguire, J. R., Hartl, C., Philippakis, A. A., del Angel, G., Rivas, M. A., Hanna, M., McKenna, A., Fennell, T. J., Kernytsky, A. M., Sivachenko, A. Y., Cibulskis, K., Gabriel, S. B., Altshuler, D., and Daly, M. J. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat. Genet. 43* (May 2011), 491–498.

[14] Ewing, B., and Green, P. Base-calling of automated sequencer traces using phred. II. Error probabilities. *Genome Res. 8* (Mar 1998), 186–194.

[15] Ferragina, P., Giancarlo, R., and Manzini, G. The engineering of a compression boosting library: Theory vs practice in bwt compression. In *ESA 2006* (2006), pp. 756–767.

[16] Ferragina, P., Giancarlo, R., Manzini, G., and Sciortino, M. Boosting textual compression in optimal linear time. *J. ACM 52*, 4 (2005), 688–713.

[17] Ferragina, P., and Manzini, G. Compression boosting in optimal linear time using the burrows-wheeler transform. In *SODA 2004* (2004), pp. 655–663.

[18] Hach, F., Hormozdiari, F., Alkan, C., Hormozdiari, F., Birol, I., Eichler, E. E., and Sahinalp, S. C. mrsFAST: a cache-oblivious algorithm for short-read mapping. *Nature Methods 7*, 8 (2010), 576–577.

[19] Haussler, D., O'Brien, S. J., Ryder, O. A., Barker, F. K., Clamp, M., Crawford, A. J., Hanner, R., Hanotte, O., Johnson, W. E., McGuire, J. A., Miller, W., Murphy, R. W., Murphy, W. J., Sheldon, F. H., Sinervo, B., Venkatesh, B., Wiley, E. O., Allendorf, F. W., Amato, G., Baker, C. S., Bauer, A., Beja-Pereira, A., Bermingham, E., Bernardi, G., Bonvicino, C. R., Brenner, S., Burke, T., Cracraft, J., Diekhans, M., Edwards, S., Ericson, P. G., Estes, J., Fjelsda, J., Flesness, N., Gamble, T., Gaubert, P., Graphodatsky, A. S., Marshall Graves, J. A., Green, E. D., Green, R. E., Hackett, S., Hebert, P., Helgen, K. M., Joseph, L., Kessing, B., Kingsley, D. M., Lewin, H. A., Luikart, G., Martelli, P., Moreira, M. A., Nguyen, N., Orti, G., Pike, B. L., Rawson, D. M., Schuster, S. C., Seuanez, H. N., Shaffer, H. B., Springer, M. S., Stuart, J. M., Sumner, J., Teeling, E., Vrijenhoek, R. C., Ward, R. D., Warren, W. C., Wayne, R., Williams, T. M., Wolfe, N. D., and Zhang, Y. P. Genome 10K: a proposal to obtain whole-genome sequence for 10,000 vertebrate species. *J. Hered. 100* (2009), 659–674.

[20] Hormozdiari, F., Hach, F., Sahinalp, S. C., and Alkan, C. Sensitive and fast mapping of di-base encoded reads. *Bioinformatics 27*, 14 (2011), 1915–1921.

[21] Hsi-Yang, F. M., Leinonen, R., Cochrane, G., and Birney, E. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res. 21* (May 2011), 734–740.

[22] Huffman, D. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (sept. 1952), 1098 –1101.

[23] Kodama, Y., Shumway, M., and Leinonen, R. The sequence read archive: explosive growth of sequencing data. *Nucleic Acids Res* (Oct 2011). Epub ahead of print.

[24] Kozanitis, C., Saunders, C., Kruglyak, S., Bafna, V., and Varghese, G. Compressing genomic sequence fragments using SlimGene. In *RECOMB 2010* (2010), pp. 310–324.

[25] Li, H., and Durbin, R. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics 25*, 14 (2009), 1754–1760.

[26] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G. T., Abecasis, G. R., and Durbin, R. The sequence alignment/map format and SAMtools. *Bioinformatics 25*, 16 (2009), 2078–2079.

[27] Mäkinen, V., Navarro, G., Sirén, J., and Välimäki, N. Storage and retrieval of individual genomes. In *RECOMB 2011* (2009), pp. 121–137.

[28] Miller, R. E., and Thatcher, J. W., Eds. *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York* (1972), The IBM Research Symposia Series, Plenum Press, New York.

[29] Pearson, W. R., and Lipman, D. J. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences 85*, 8 (1988), 2444–2448.

[30] Rissanen, J., and Langdon, G. G. Arithmetic coding. *IBM Journal of Research and Development 23*, 2 (march 1979), 149 –162.

[31] Sahinalp, S. C., and Vishkin, U. Symmetry breaking for suffix tree construction. In *STOC 1994* (1994), pp. 300–309.

[32] Sahinalp, S. C., and Vishkin, U. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *FOCS 1996* (1996), pp. 320–328.

[33] Said, A. *Introducing to Arithmetic Coding - Theory and Practice*, published as a chapter in lossless compression handbook by khalid sayood ed. HPL-2004-76. Imaging Systems Laboratory, HP Laboratories Palo Alto, Apr. 2004.

[34] Schadt, E. E., Linderman, M. D., Sorenson, J., Lee, L., and Nolan, G. P. Computational solutions to large-scale data management and analysis. *Nat. Rev. Genet. 11* (Sep 2010), 647–657.

[35] Sherry, S. T., Ward, M. H., Kholodov, M., Baker, J., Phan, L., Smigielski, E. M., and Sirotkin, K. dbSNP: the NCBI database of genetic variation. *Nucleic Acids Res. 29* (Jan 2001), 308–311.

[36] Tembe, W., Lowey, J., and Suh, E. G-sqz: compact encoding of genomic sequence and quality data. *Bioinformatics 26*, 17 (2010), 2192–2194.

[37] Wan, R., Anh, V. N., and Asai, K. Transformations for the compression of fastq quality scores of next-generation sequencing data. *Bioinformatics 28*, 5 (2012), 628–635.

[38] Witten, I. H., Neal, R. M., and Cleary, J. G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June 1987), 520–540.

[39] Yanovsky, V. ReCoil - an Algorithm for Compression of Extremely Large Datasets of DNA Data. *Algorithms Mol Biol 6* (Oct 2011), 23.

[40] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (1977), 337–343.

[41] Ziv, J., and Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*, 5 (1978), 530–536.