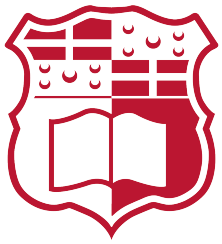
A MapReduce Approach to Genome Alignment

Karl Pullicino

Supervisor: Dr Jean-Paul Ebejer



Department of AI

Faculty of ICT

University of Malta

September 2017

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Artificial Intelligence

L-Università ta' Malta

FACULTY/INSTITUTE/CENTRE/SCHOOL ICT

DECLARATIONS BY POSTGRADUATE STUDENTS

Student's I.D. /Code 319188M

Student's Name & Surname Karl Pullicino

Course M.Sc. A.I.

Title of Dissertation A MapReduce Approach to Genome Alignment

(a) Authenticity of Dissertation

I hereby declare that I am the legitimate author of this Dissertation and that it is my original work.

No portion of this work has been submitted in support of an application for another degree or qualification of this or any other university or institution of higher education.

I hold the University of Malta harmless against any third party claims with regard to copyright violation, breach of confidentiality, defamation and any other third party right infringement.

(b) Research Code of Practice and Ethics Review Procedures

I declare that I have abided by the University's Research Ethics Review Procedures.

As a Master's student, as per Regulation 58 of the General Regulations for University Postgraduate Awards, I accept that should my dissertation be awarded a Grade A, it will be made publicly available on the University of Malta Institutional Repository.

Signature of Student

KARL PULLICINO
Name of Student (in Caps)

14/05/2018
Date

Faculty of ICT

Declaration

I, the undersigned, declare that the dissertation entitled:

A MapReduce Approach to Genome Alignment

submitted is my work, except where acknowledged and referenced.

Karl Pullicino

29 September 2017

Acknowledgements

Several people provided continuous support and help throughout the duration of this dissertation. I start with a big thank you to my family for their support and help through all my life, encouraging me even when times were not so bright. I would like also to thank several friends who were always there for me, providing support, help, good company and comfort during dark times.

Special mention goes to my supervisor Dr Jean-Paul Ebejer for his continuous guidance and help all the way from the start till the end of the dissertation.

This work is supported by the University of Malta by funding the costs incurred in cloud computing on both AWS and Microsoft Azure.

Abstract

Recent years brought an enormous growth in DNA sequencing capacity and speed, thanks to the application of Next-Generation Sequencing (NGS) technologies. The alignment of read sequences to a given reference genome is crucial for further diagnostic downstream analysis. Finding the optimal alignment of short DNA reads from a biological sample to a reference human genome, requires big data techniques, since reads' size are in the region of 200GB. In this dissertation we present two approaches to perform distributed sequence alignment of genomic data based on the MapReduce programming paradigm. `MR-BWA` presents a novel approach in distributing BWA in a different manner than existing work. BWA is an industry standard software used for genomic reads alignment. `MR-BWT-FM` presents low level optimizations on suffix array and BWT creation which are used to create a custom FM-Index which in turn is used for distributed genome sequence alignment. Output generated by the application generates insights and charts about the results. We evaluate the performance and correctness of both approaches by comparing our output with that of similar tools, using standard datasets from the 1000 Genomes Project. Performance and correctness results for both distributed approaches are comparable with similar tools, whilst the final custom FM-Index size is smaller than the standard BWA index size. The source code of the software described in this dissertation is publicly available at `https://github.com/kpullu/msc`.

Keywords: DNA; genomics; sequence alignment; suffix array; bwt; fm-index; MapReduce; big data; cloud computing

Contents

List of Figures

List of Tables

List of Acronyms

BWA Burrows-Wheeler Aligner.

BWA-SW Burrows-Wheeler Aligner Smith-Waterman.

BWASW-PMR Burrows-Wheeler Aligner Smith-Waterman Alignment on Parallel MapReduce.

BWT Burrows-Wheeler Transform.

BWT-FM Burrows-Wheeler Transformation with FM-Indexing.

CG Complete Genomics.

DAWG Directed Acyclic Word Graph.

DNA Deoxyribonucleic Acid.

EBI European Bioinformatics Institute.

EMR Elastic Map Reduce.

FM-Index Ferragina and Manzini Index.

GB Gigabytes.

GO Gene Ontology.

HDFS Hadoop Distributed File System.

JNI Java Native Interface.

LF Last First.

MPI Message Passing Interface.

MR-BWA MapReduce Burrows-Wheeler Aligner.

MR-BWT-FM MapReduce Burrows-Wheeler Transformation with FM-Indexing.

NGS Next-Generation Sequencing.

NW Needleman-Wunsch.

PPI Protein-Protein Interaction.

PT Prefix Trie.

SA Suffix Array.

SAM Sequence Alignment/Map.

SRA Short-Read Alignment.

SW Smith-Waterman.

TB Terabytes.

TSV Tab Separated Values.

1. Introduction

The U.S. National Library of Medicine (2016) [1] states that "Deoxyribonucleic Acid (DNA), is the hereditary material in humans and almost all other organisms. The information in DNA is stored as a code made up of four chemical bases: adenine (A), guanine (G), cytosine (C), and thymine (T)". This is shown in Figure 1.1. The human DNA is made up of approximately three billion of these bases. Individuals have more than 99% of identical bases. In recent years, Next-Generation Sequencing (NGS) technologies brought an enormous growth in DNA sequencing capacity and speed [2]. These led to a tremendous growth in the amount of short-read sequence datasets. Short reads are used since NGS technologies are limited and cannot sequence the whole human genome at once. Aligning produced reads from NGS technologies to a reference genome, Short-Read Alignment (SRA), is an essential operation required for further analysis. An alignment defines the arrangement of multiple reads in order to reconstruct the DNA sequence.

1.1 Motivation

One of the main computational problems in bioinformatics [3] is finding the optimal way of aligning many short DNA reads from a biological sample to a reference human genome so that the DNA of the sample can be assembled. The underyling chemistry of NGS cannot sequence the whole human genome at once, hence short-

read overlapping sequences are used. This means that each base pair in the genome is sequenced a number of times, for example having each base sequenced on average 30 times. For this reason, NGS produce up to six billion reads per run. Such considerable amount of data requires approximately four days to be processed on a single 16-core machine [4]. Therefore, scientists require to have scalable solutions which improve the aligners' performance and hence obtaining results in practical time. Cloud computing services are the most viable solutions to perform large-scale on-demand analysis. Using cloud platforms, one can solve the storage and computing problems that exist in genome alignment. Such alignments and related datasets are available on the cloud ready to be used by bioinformaticians [1] [2] [3]

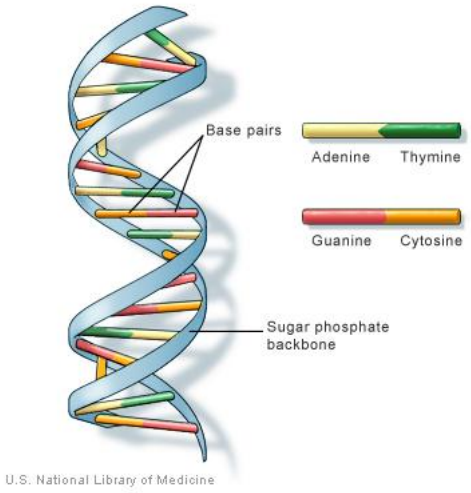


Figure 1.1: DNA Structure [1]

1.2 DNA Alignment Problem

DNA sequencing technology suffers from experimental errors and therefore there are likely to be errors in the produced DNA sequence. Each base in the genome is sequenced multiple times in order to compensate for errors that might occur.

[1] `http://www.internationalgenome.org/data/`
[2] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/`
[3] `https://aws.amazon.com/1000genomes/`

This is defined as coverage. Having a 30-fold coverage indicates that each base was sequenced on average 30 times. Currently, for every three billion bases representing the human genome, 30-fold additional data ($\approx$ 100 gigabases) is being collected to cater for sequencing errors [5]. Chen [6] shows how short reads are aligned with reference genome in Figure 1.2. Currently one sample is approximately 200GB whilst between 2 and 40 exabytes will be needed by 2025 only for the human genomes [5]. Obviously this leads to extensive research of data compression and minimizing as much as possible the need to oversample by using longer reads. Effectively, the more times, the same section of DNA is sequenced/read, the more confidence there is that the final sequence is correct. Although coverage and accuracy in NGS technologies increased over the years, they still have a high error rate. Having a higher coverage reduces the likelihood of there being gaps in the final assembled sequence, however, this results in having lots of DNA reads to assemble, hence increasing the dataset size. The sequencing machine generates a large number of reads from the sample. Each read is a randomly located segment of the DNA sequence, having lengths in the order of 100 to 1000 base pairs, depending on the sequencing technology being used. The number of reads is the range between 10s to 100s of millions. The *DNA assembly problem* is defined as being able to reconstruct the DNA sequence from the many short reads [7]. Longer reads are better because are more likely to anchor repetitive sequences with non-repetitive sequences hence making them distinguishable.

1.3 Aims and Objectives

The main aim of this dissertation is to provide a cloud-based implementation performing distributed sequence alignment of genomic data by building up upon already existing applications. The MapReduce [12] framework model and Apache Hadoop are used as underlying technology, whilst industry standard software Burrows-Wheeler Aligner (BWA) [9] is a main building block for our system. Our

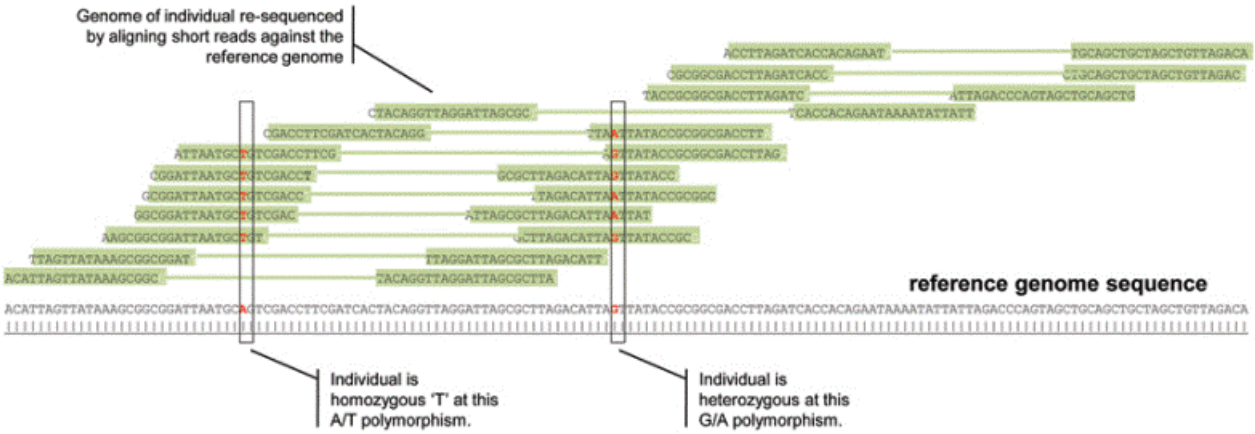


Figure 1.2: Aligning short reads to the reference genome [6]

approach extends on the work done in [4, 8]. From a technical perspective, our first and foremost aim is to optimize the sequence alignment through distribution and optimized indexing techniques which reduce index disk size. This is obtained through the following objectives:

- Parallelization of Smith-Waterman (SW) algorithm [10] for genomic reads alignment
- Sequence alignment distribution on MapReduce [12] by enabling distributed map and reduce workers processing reads.
- Optimization of Suffix Array (SA), keeping in consideration repetitive sequences, which drastically reduce index size and lookup time

Eventually, the resultant alignments are used to reassemble the human genome from a biological sample and compare it with the reference human genome. Alignment sample correctness is evaluated by aggregating all read queries base pairs alignment positions and compare them with the reference genome position. The output generated by the application should generate insights about the results. The end implemented software should be easily maintainable.

The first objective, relating to the parallelization of the Smith-Waterman algorithm, was unsuccessful since our implementation suffered from out of memory

errors as explained in Section 3.4. For this reason, we had to change tack and our objectives became the implementation of a distributed aligner based on BWA and optimization of Suffix Array.

1.4 Big Data in DNA Sequencing

The first sequencing technology knows its roots to Frederick Sanger et al. in 1977 [15]. They DNA sequencing technology developed at that time was based on chain-termination method, which is sometimes also referred to as Sanger sequencing. The human genome project in 2001 would not have been completed without the Sanger sequencing technology [16]. This human reference genome which contains approximately 3 billion base pairs, took 10 years to be completed and costed nearly three billion US dollars. This sped up NGS technologies development. NGS technologies improve upon the Sanger method with respect to substantial parallel analysis, accuracy and reduced cost as seen in Figure 1.3

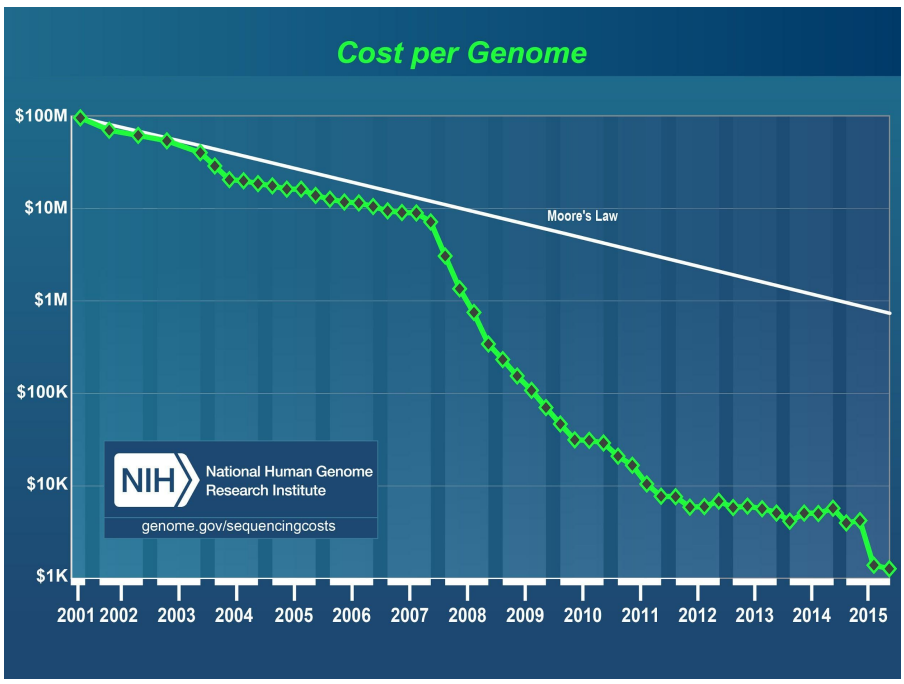


Figure 1.3: Cost per genome. 2007 sees a sharp drop due to the introduction of NGS technologies [19]

Data analysis for NGS genome sequences is the main bottleneck due to the huge amount of data involved [2]. A fundamental feature in NGS is DNA polymerase chain reaction. Basically given a single strand template and base, it builds a double stranded copy of the template by simply knowing a little bit of the sequence around the desired region. Also, a technique called base caller is used to deal with ambiguity. This reports the base quality for each base call. This is defined as the estimate of the probability that the base was called incorrectly. Figure 1.4 outlines the exponential growth in bioinformatics data brought by NGS. Reads' size for a single human is approximately 200 gigabytes [17]. In 2014, the European Bioinformatics Institute (EBI) had approximately 40 petabytes of data about genes, proteins, and small molecules [18].

A recent study by Stephens et al. [5], classified genomics as the most demanding domain in terms of data acquisition, storage, distribution, and analysis, when compared with three other Big Data domains: astronomy, YouTube and Twitter. These four big data domains were compared based on the four main elements that make up the life cycle of a dataset: acquisition, storage, distribution, and analysis.

Genomics data acquisition is highly distributed and heterogeneous. Stephens et al. [5] claim that sequencing data rate of growth has been doubling every seven months over the past decade. US and China both aim to sequence 1 million genomes in the next few years. Having the world's population estimated to be approximately eight billion by 2025, it is possible that 25% of the population in developed countries will have their genomes sequenced.

Instead of using conventional laboratories, biologists nowadays rely on huge genomic data made available by various research groups. Analyzing such an enormous amount of data and its storage are computational problems. Currently, NGS technologies, as is the Illumina HiSeqX Ten, generate approximately six billion reads per run. Processing this enormous amount of data on a single 16-core machine using BWA, takes roughly four days [4]. Therefore, NGS professionals require scalable technologies to align genomic data in reasonable time. Cloud computing

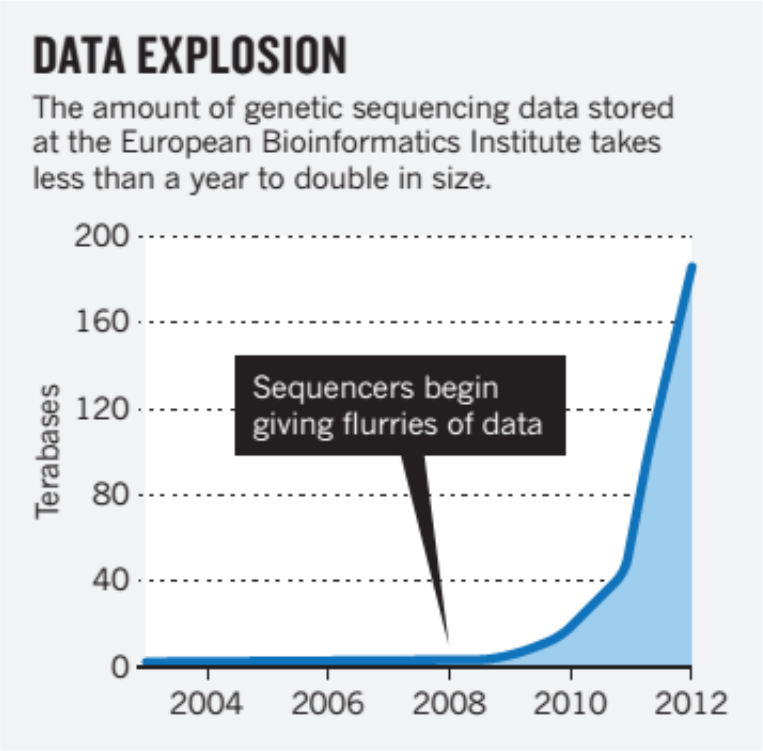


Figure 1.4: Growth of genetic sequencing data. 2008 shows a sharp rise due to the introduction of NGS technologies [20]

services provide the most practical solution to perform huge genomic data analysis. Calabrese and Cannataro [21] specify the four main advantages of such cloud model: service-oriented, massive scalability, on-demand resource and virtualization. Cloud users pay for the utilized storage and computing resources without worrying about maintenance, availability, and reliability related issues. In genomic analysis research, the main cloud computing providers are Google Cloud Genomics [4], Amazon [5], and Microsoft Azure [6]. However, in addition to having genomic applications on the cloud, new methods of data reliability and security are required to ensure privacy. A serious medical data breach would have a catastrophic effect. However, certain current encryption systems are too computationally expensive for widespread use.

Although cloud computing provides scalable and flexible infrastructure, parallel

[4] `https://cloud.google.com/genomics/`
[5] `https://aws.amazon.com/health/genomics/`
[6] `https://azure.microsoft.com/en-us/`

computing models on cloud platforms/infrastructure are required to analyze genome sequences. These are needed since genomic data analysis involves intensive and computationally complex stages. A usecase brought up by Langmead explains how variant calling with two billion genomes per year and using parallel computation involving 100,000 CPUs, requires methods three-to-four times faster than current capabilities [22]. A commonly used framework for the cloud, is called MapReduce [12]. This is explained in more detail in Section 2.2.

1.5 Document Structure

The rest of the document is organized as follows. In Chapter 2, technical background about DNA alignment is given. Moreover the MapReduce programming paradigm is explained in more detail. A review of applicable string matching algorithms used in genome alignment, is presented. Related work to this dissertation is described at the end of Chapter 2, highlighting already existing distributed techniques. Chapter 3 provides the reader with a detailed overview of the design and implementation of our approaches. A general overview of the system is first presented including the application's input and output. The main MapReduce functions used in our approaches are explained and the overall system architecture is presented. We justify decisions taken and optimizations over existing work, by explaining the design and implementation for each approach. Chapter 4 presents the results obtained using our approaches. A literature comparison with a similar application is performed, in terms of speed. Moreover, we present our approach for evaluating alignment correctness. Finally, this dissertation concludes with an explanation of possible advancements and pitfalls of our system. Future improvements are also presented. An appendix explaining how to set up and use the application, is also available.

2. Background and Literature Review

Sequence alignment between sample reads and reference genome is a widely used procedure in bioinformatics in order to determine similarity. Various algorithms have been utilized in order to tackle the DNA alignment problem. These include both dynamic programming based approaches and probabilistic based methods. Dynamic programming algorithms ensure accurate and optimal alignment, however, due to the memory required for larger alignments, such methods are slow and limited to only processing short sequences. In order to address high computation needs for long alignments, cluster computing platforms have been provided to researchers. However, these are constrained by hardware capacity and concurrent access capacity to support multiple users. This brought a divergence between the available computing capabilities and the sequencing throughput needed. For this reason, cloud computing platforms comes to rescue since they provide scalability, and on-demand access to computing resources offering cost effective processing of Terabytes scale data. In this section we will go through basic theory about DNA alignment and what algorithms apply for such task. Also, big data paradigms used in relation to string algorithms are explained, with reference to existing work.

2.1 DNA Alignment and Assembly

The resulting DNA sequence produced from genome sequencing technology contains inherent experimental errors. To make up for such errors, each base in the genome is sequenced multiple times; this defines coverage. For example, 30 times (30-fold) coverage means each base is sequenced on average 30 times. Langmead [24] defines the overall average coverage as $\frac{total\,length\,of\,all\,reads}{total\,length\,of\,genome}$. Putting reads together in the correct order to construct the complete genome sequence and identifying areas of interest is done through two processes; alignment and assembly. Sequence alignment is when a new DNA sequence is compared to existing DNA sequences to find any similarities or discrepancies between them and then arranged to show such features. vlab.amrita.edu (2012) state that sequence alignment "is used to infer structural, functional and evolutionary relationship between the sequences" [25]. Alignment of sequences is biologically important to be able to discover genetic variants which may cause diseases. De novo genome assembly is used when a reference genome is not available [27]. It involves taking a large number of DNA reads, looking for areas in which they overlap with each other and then gradually piecing them together. It is an attempt to reconstruct the original genome. In *Algorithms for DNA Sequencing*, Langmead [24] explains three laws of assembly:

1. If a suffix of read A is similar to a prefix of read B, then A and B might overlap in genome.

2. More coverage leads to more and longer overlaps.

3. Repeats makes assembly difficult. In fact approximately half of the human genome is made up of repetitive DNA sequences.

However, sequencing differences are legitimate. Two main reasons being; sequencing errors and polyploidy. Sequencing errors can occur from pipetting mistakes during the sequencing reaction e.g. gel loading or poor lane tacking. Polyploidy occurs because humans have two copies of each chromosome, one from each parent, hence

difference is real.

When we sequence an individual sample from a species that we already have a reference genome for, the new generated reads are then aligned with the reference genome to find out variations. This means that DNA reads produced after sequencing are compared to the reference genome and mapped to their most similar counterpart. A reference genome is generally long with repetitive elements whilst reads are short in length, typically, 50 to 150 base pairs. The International SNP Map Working Group [28] explain how dissimilarities in DNA sequence acquired through inheritance lead to phenotype variation. Such differences influence an individual's characteristics leading to a risk of disease. Uses of sequencing include finding rare genetic diseases, forensics, evolution, studying tumors so as to find treatments, studying microbes and bacteria in our body.

Given the huge data size involved in DNA sequence, big data paradigms are needed in the bioinformatics field in order to analyze the resultant sequence. Due to the huge data size involved in sequence alignments, data cannot be easily moved. Hence, instead of moving the data to the computation, we move the computation code to perform the analysis. A commonly used framework for massive computing in the cloud is MapReduce [12].

2.2 MapReduce

MapReduce is a distributed computing framework created by Google, defined as the key technology for processing large datasets on a cluster made by over one thousand commodity machines [12]. Apache Hadoop is one of the main used open-source implementations of MapReduce.

Sudha and Baktavatchalam (2010) define MapReduce as "a big data programming paradigm that expresses a large distributed computation as a sequence of distributed operations on data sets of key/value pairs" [29]. The two main functions of a MapReduce program are: `map, reduce`. The map function processes input

data and generates a list of `<key, value>` pairs. The input data is first split into a number of chunks distributed across nodes. Each chunk is assigned to a map task. In this case, the framework is responsible for distributing these map tasks across data nodes in the cluster on which it operates as shown in Figure 2.1. The intermediate transient data outputted by the map tasks is shuffled and sorted by keys. In the shuffle and sort phase, data from all mappers is sorted in order to be grouped by the key. Eventually data is forwarded to the reducers. The crucial part, is that all values associated with the same key go the same single reducer. Input to the reduce function are sorted `<key, [values list]>` pairs. It aggregates the results having the same key into `<key, value>` pairs and writes the output data to disk. Sometimes, an optional phase called `combine` is also used. This phase runs between the map phase and the shuffle/sort phase in order to aggregate key/value pairs outputted from the mapper which have the same key. The `combine` function helps in reducing network traffic, however, its execution is optional and Hadoop can decide not to execute a particular combiner. MapReduce has a master/slave architecture. The master node, also called JobTracker, is aware of the data available on each node/slave in the cluster. The main responsibility of the JobTracker is to schedule the MapReduce tasks over the slave nodes in the most efficient way possible depending upon the available input dataset. This helps in avoiding unnecessary data transfer. The slave nodes are called TaskTrackers, which execute MapReduce tasks in parallel.

The other principal component making up Hadoop is, Hadoop Distributed File System (HDFS). HDFS provides a scalable and reliable data storage working across large clusters of commodity hardware. The architecture of HDFS is highly fault-tolerant due its data replication and distribution. Each file is stored in sequential blocks of same size, although the size of the block is configurable. The files are replicated (by default having three replicas) over multiple machines to avoid data loss over node failures and provides high aggregate throughput for streaming large files. HDFS is also designed in a master/slave architecture. An HDFS cluster is

made up of a single NameNode and multiple DataNodes. In this case, the NameNode acts as a master file server whereby it manages the distributed file system and acesses to all files. This makes the NameNode a single point failure and hence, HDFS does not provide high availability. The DataNodes are responsible for read and write operations over the files[1].

Hadoop suffers when iterative applications are executed in the framework and multiway joins are needed. Moreover, Hadoop framework, first performs sequential processing of all the mapping phases and eventually performs the reduce stage. This affects badly the performance. An overview of Hadoop MapReduce execution framework is shown in Figure 2.1.

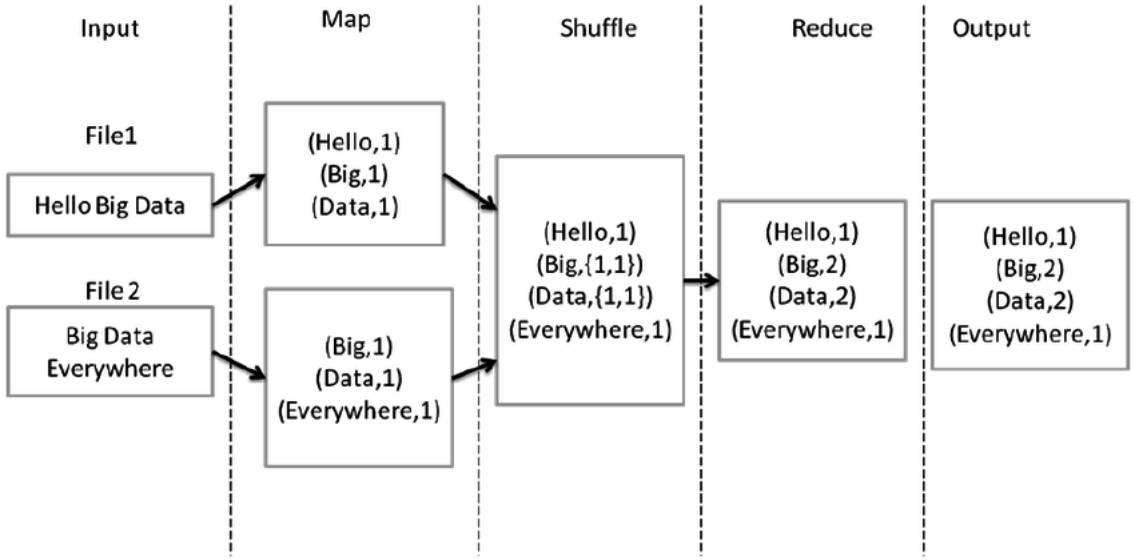


Figure 2.1: Hadoop MapReduce execution framework showing a word count example [30]

2.3 String Matching Algorithms and Indices

Given that the human genome is represented as a string of three billion character (bases A, C, G, T), the problem at hand is to find the most effective string matching

[1] `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`

algorithms, using optimized indices in order to align the input sequences with the reference genome.

2.3.1 Naive-Exact Matching Algorithm

The basic approach of matching a pattern with a text, is by using a naive exact matching algorithm. The worst case for this approach is $x(y - x + 1)$ where x is the length of each read and y is the length of the reference genome. $y - x + 1$ is defined as every position in y where x can start. This happens where every character comparison results in a match. This is definitely impractical in case of matching genomic reads with a reference genome since we need to go through the whole reference genome for each read. Moreover, naive-exact matching would not work since reads have mismatches to reference genome due to sequencing errors and/or insertions and deletions (indels). Hence this approach is not feasible.

2.3.2 Boyer-Moore Algorithm

The Boyer-Moore algorithm [31] extends the naive approach. Comparisons are performed right to left and repetitive alignments are skipped. In case of a mismatch (or a complete match of the whole pattern) two pre-computed functions are used. These are:

- Bad-character rule: upon mismatch, alignments are skipped until a mismatch becomes a match or pattern moves past mismatched character, that is, it turns a mismatch into a match. This is the ideal case as the pattern can be shifted to the mismatch position as shown in Figure 2.2

- Good-suffix rule: skips until there are no mismatches between pattern and substring matched by inner loop or else pattern has moved past matched substring, that is, tries keep match as match without turning to mismatch. This is the less ideal case as shown Figure 2.3. Simha [33] describes the rule required at each step:

1. Start comparing from the right end of the pattern
2. Identify the mismatch character in the text and its position
3. Find the rightmost occurrence of this character in the pattern, that is to the left of the mismatch position
4. Move the pattern rightwards to align these two characters

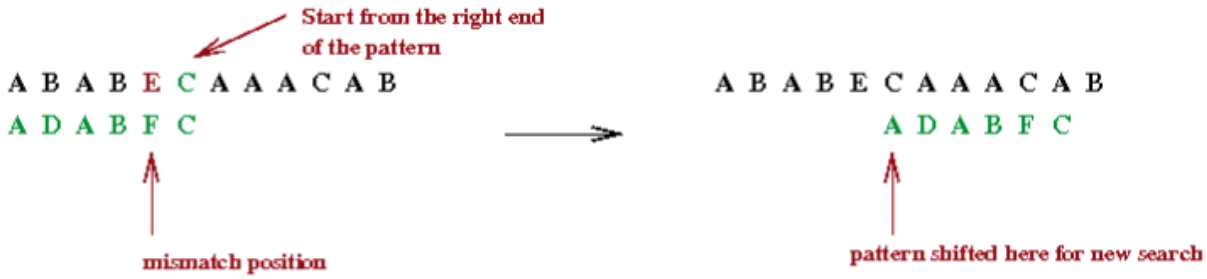


Figure 2.2: Boyer-Moore bad-character rule [33]

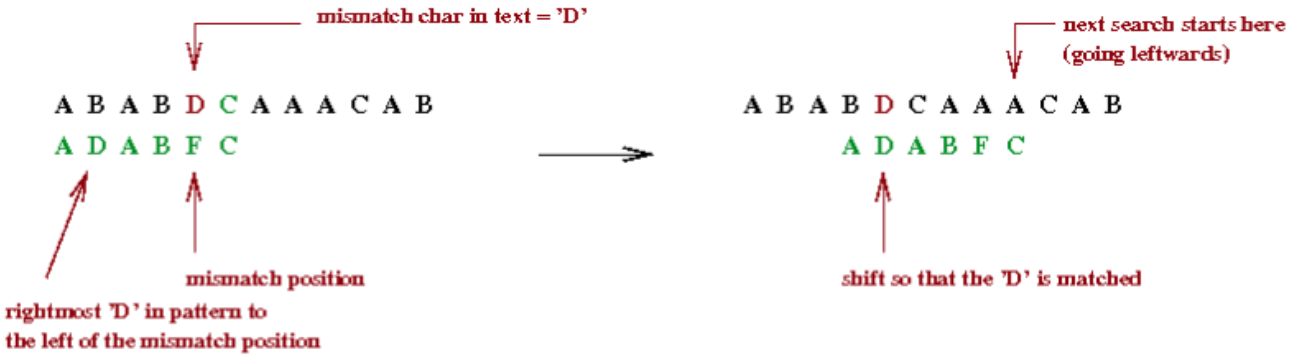


Figure 2.3: Boyer-Moore good-suffix rule [33]

Boyer-Moore builds lookup tables for both rules based on pattern before performing alignment. These lookup tables are built using the Z-algorithm as defined by Gusfield [32]. This algorithm finds all occurrences of a pattern in a text in linear time. The worst-case running time is still $O(mn)$ but it is faster than the naive-exact matching approach.

However, this means, that in case of mapping reads we need to build lookup tables for each different read. This would be impractical since NGS generates

millions of reads. Hence, we need an offline solution, that is, index and pre-process reference genome once since this this does not change across runs, then compare each read to the indexed reference genome.

2.3.3 K-Mer Index

Indexing the whole reference genome, requires efficient string indexing. A *k-mer* index refers to the scenario where each substring of length k is indexed. After creating the index, all the offsets where a pattern P occurs within text T can be found. This consists of two steps; finding the index hits, that is, the place where the *k-mer* occurs within the text, that is the first step. Then, second step is verification, that is, determining if there is a full match of P to T. Generally a hashtable data structure is used for such index. Optimizations on *k-mer* index include indexing every other *k-mer* instead of indexing all *k-mers*, e.g., indexing only even *k-mers*. Then querying is done using both even and odd offsets with respect to pattern P. In general, every nth *k-mer* is taken. Another optimization on the *k-mer* index is by using subsequence instead of substring. The main difference is that subsequence of a string S is a string consisting of characters that also occur in S, in the same order, but not necessarily consecutively, which is a requirement of a substring. The substring is a refinement of the subsequence, this makes the index filter more specific. For example, having string *banana*:

- *ana* is both a substring and a subsequence
- *anna* is only a subsequence

2.3.4 Suffix Tree and Suffix Array

A suffix tree is used for exact string matching and also to find the longest common substring of two strings. However this takes $O(n^2)$ to build, which is reduced to $O(n)$ time with Ukkonen algorithm [34]. However, n may be large and the index consumes a lot of space. A suffix index has $\frac{n(n+1)}{2}$ characters which makes it impractical. An

improvement on suffix tree, is Suffix Array (SA) since this grows linearly. This is done by traversing a suffix tree in depth-first-search lexicographically, picking edges and filling the suffix array as outlined in part **b** of Figure 2.4. Given a text S of length n, the suffix array for S, is an array of integers of range 1 to n specifying the lexicographic ordering of the suffixes of the string S. Each entry in the SA specifies the starting position of a specific suffix in a string. Once the SA has been built, it can be queried using various search algorithms. This takes $O(n)$ time. Searching in the suffix array is done through a binary search and this takes $O(m\,logn)$ time. Manber and Myers [35] propose a further optimisation on this by using an algorithm called prefix doubling. At step k, the orders of the partial prefixes is $s[i : i + k]$. But $s[i : i + 2 * k] == s[i : i + k] + s[i + k : 2 * k]$ so the order and the shifted order can be merged at each step to obtain the new order. After at most $log(n)$ steps, the prefixes are totally sorted.

2.3.5 Burrows-Wheeler Transform (BWT)

Burrows-Wheeler Transform (BWT) is defined by Langmead [79] as a way of permuting the characters of a string T into another string $BWT(T)$. It was first presented as a compression technique [55]. BWT brings identical characters together into runs since sorting is done according to right-context. This makes it more compressible since run-length encoding can be use for repetitive strings. The BWT is similar to suffix array, since in BWT, text T rotations are sorted, whilst in SA, text T suffixes are sorted. However sort order for rotations and suffixes is the same. In fact BWT can be defined as *"characters just to the left of the suffixes in the suffix array"* (Langmead, 2013) [79]. This can be seen more formally in Equation 2.1:

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{if SA[i]} > 0 \\ \$ & \text{if SA[i]} = 0 \end{cases} \tag{2.1}$$

An important feature in BWT is the Last First (LF) columns mapping. This states that the ith occurrence of a character c in L has the same rank as the ith occurrence of a character c in F. Occurrences of c ranks appear in same order in F and L. This happens because they are sorted by right-context in both F and L. With B-ranking, F has a very simple structure; start with \$, then blocks of characters in ascending order e.g. in case of DNA, we would have blocks of As, followed by Cs, Gs and Ts as shown in part **a** of Figure 2.4. This makes it ideal for DNA alignment due to its repetitive structure and small alphabet. Another feature of BWT is that it is a reversible permutation. Recreating T from $BWT(T)$ is done by starting in first row and apply LF repeatedly accumulating predecessors along the way.

2.3.6 FM-Index

The Ferragina and Manzini Index (FM-Index) is defined as "a hybrid of BWT and suffix arrays" (Berger, Peng & Singh, 2013) [36]. Ferragina and Manzini [37] describe how the BWT, together with other auxiliary data structures, can be used to create an index of T which is space-efficient. The core of the FM-Index are the F and L from BW Matrix. Applying the LF mapping for a number of times extracts the rows prefixed by the suffixes of P. The number of rows extracted equates to the number of times P occurs in T (if no rows are extracted then P does not occur in T). All suffixes of the original sequence are cound in the FM-Index. This allows for faster subsequence mapping in $O(m)$ time [36]. Moreover, having a genome of length n requires only $O(n)$ space for the FM-Index. Berger et al. state that "these features make the FM-Index ideal for short-read mapping, where the read length m is usually small, and the size of the reference genome is large" (Berger, Peng & Singh, 2013) [36].

Choosing the correct indexing for the human reference genome (which has $3 * 10^9$ characters) is crucial, since as explained by Langmead and Pritt [24] this leads to a drastic difference in size:

1. Suffix Tree $\geqslant$ 45GB
2. Suffix Array $\geqslant$ 12GB
3. BWT / FM-Index $\geqslant$ 1GB

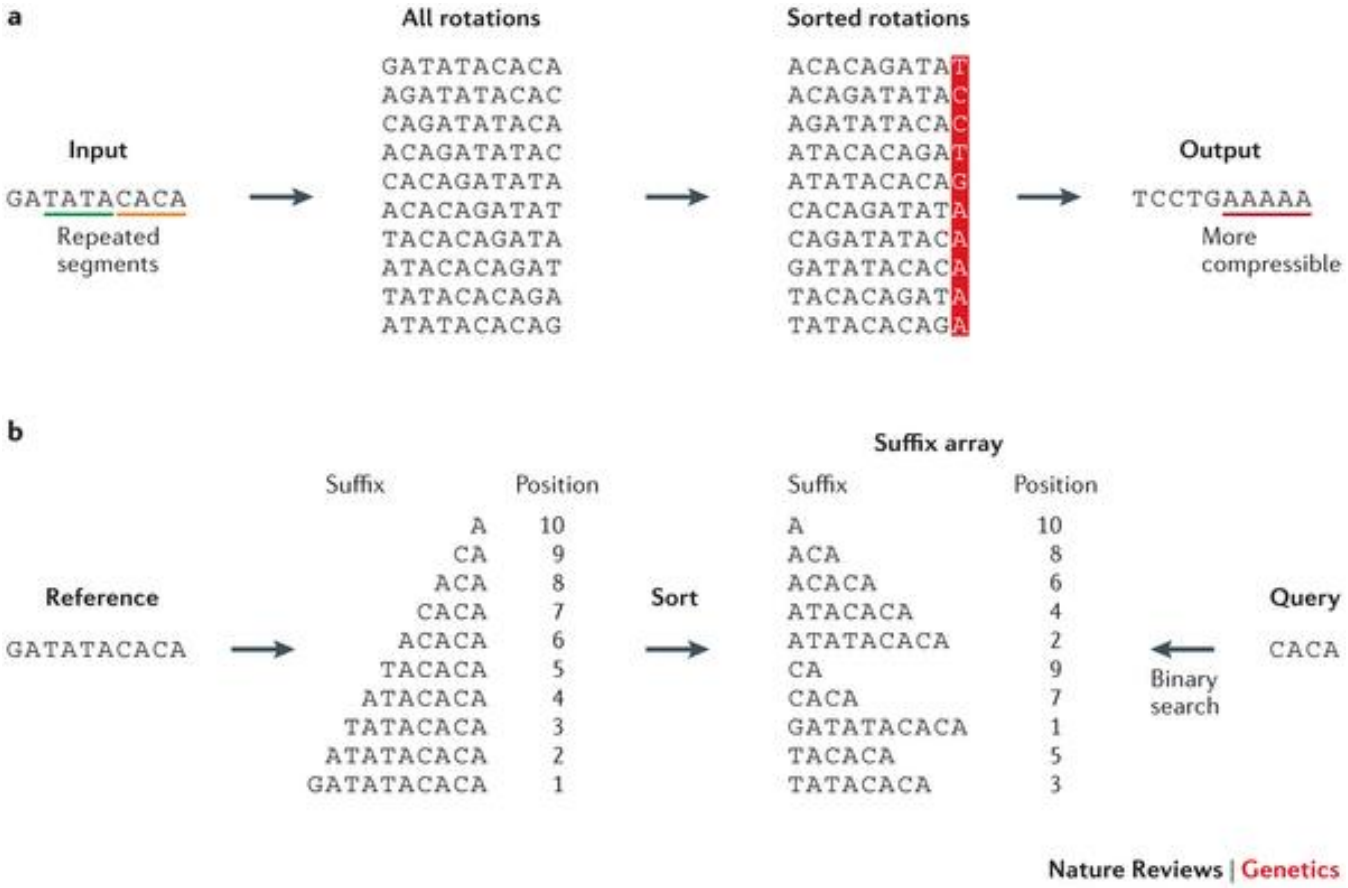


Figure 2.4: BWT and Suffix Array. Part (**a**) shows an example of BWT. Basically $BWT('GATATACACA') = 'TCCTGAAAAA'$. Part (**b**) shows suffix array for input $'GATATACACA'$ [36]

2.3.7 Timsort

Timsort [38] is based on the idea that in the real world, the input data array to be sorted contains ordered (ascending or descending) sub-arrays. This makes it an adaptive sorting algorithm that benefits from already sorted input data or data which has limited amount of disordered items. It is based on three steps:

1. Split input array into sub-arrays
2. Sort each sub-array
3. The sorted sub-arrays are merged into one array using Merge Sort

Timsort is ideal in genomic data because of the repetitive structure of DNA. In the worst case, Timsort takes $O(nlogn)$ to sort n elements, whilst best-case scenario is defined as making $n-1$ comparisons.

2.3.8 Pigeonhole Principle

The Pigeonhole Principle lets you use exact matching algorithms for approximate matching. It works by dividing pattern P into partitions, checking each partition and if it matches, verification is done. Taking the case of the Boyer-Moore algorithm; using pigeonhole principle having up to n mismatches, then pattern P is divided into $n+1$ segments. If pattern P is divided into $n+1$ segments then at least one of $n+1$ segments matches perfectly. Each partition is processed through Boyer-Moore algorithm. If it matches, verification is done, else count of mismatch is kept. This approach can be used in genomic sequence alignment since there might be legitimate differences between read and reference genome due to sequencing error and/or true alterations in the subject genome. Hence, approximate matching suffices. Mismatch can occur due to indels. String distance between two strings, can be calculated through Hamming distance [61] and Levenshtein distance [62]. Hamming distance is the minimum number of substitutions to change x to y. This constraints x and y to be of the same length. Levenshtein distance is the minimum number of edits, that is, substitutions/insertions/deletions to change x to y. This allows for strings with different lengths.

2.3.9 Edit Distance

Edit distance is the formal basis of the Levenshtein distance [62], that is, the minimum number of operations required to transform one string into another. Since

we are allowing insertions and deletions, if strings x and y are of the same length, the edit distance is always smaller or equal to Hamming distance for the same two strings, that is, $editDistance(x, y) \leqslant hammingDistance(x, y)$ [39]. If strings x and y are of different length, at least a number of edits need to be introduced to make the two strings of the same length, that is, $editDistance(x, y) \geqslant \|x|\text{-}|y\|$ [39].

The principle is based on knowing the edit distance of prefixes of two strings, that is, $D[i, j]$ is the edit distance between length-i prefix of x and length-j prefix of y. Hence the edit distance is the minimum of performing a deletion on $i - 1$ or performing an insertion on $j - 1$ or performing a substitution from $i - 1$ to $j - 1$ [39]. This is formally defined in Equation 2.2:

$$Let\, D[0, j] \,=\, j,\, and\, let\, D[i, 0] \,=\, i$$

$$\text{Otherwise, let D[i,j]} = min \begin{cases} D[i-1, j] \,+\, 1 \\ D[i, j-1] \,+\, 1 \\ D[i-1, j-1] \,+\, \delta(x[i-1], y[j-1]) \end{cases} \tag{2.2}$$

$$\delta(a, b)\, is\, 0\, if\, a \,=\, b,\, 1\, otherwise$$

Implementing the above function in recursive fashion makes it extremely slow because it processes the same arguments multiple times, for example, *editDistanceRecursive('ABC','BBC')* processes *('AB','BB')* several times. This is only a three character example, let alone executing this on DNA sequences. To avoid this, the function should be run once, store it and refer to it whenever needed. This is based on memoization, that is, reusing solutions to subproblems. This leads us to dynamic programming.

2.3.10 Dynamic Programming

This approach splits the original problem into smaller pieces and finds a solutions for all sub-problems. Eventually, these are put all together, hence forming one

optimal complete solution to the whole problem. In case of alignment, the solution is found in a calculable way by assigning scores for matches and mismatches. These create scoring matrices where each cell contains edit distance between substrings and for any pair of prefixes from x, y the edit distance is calculated once. The value in a cell depends upon its upper, left, and upper-left neighbors. By searching the highest scores in the matrix, alignment can be accurately obtained. Pattern i is initialized in rows whilst text j is represented in columns whereby $D[0, j]$ is initialized to j, and $D[i, 0]$ to i. Remaining cells can be filled from top row to bottom and from left to right or vice-versa, or else opting for anti-diagonal. The backtrace mechanism is used to remember from where we came from, that is why we always need to keep reference to $i - 1$ and $j - 1$. Once the final row is reached, the full backtrace path corresponds to an optimal alignment. An optimal alignment is composed of optimal sub-alignments. Filling the matrix is $O(mn)$ space and time, and yields edit distance. The backtrace is $O(m + n)$ time and yields optimal alignment. Mathematically the number of ways of aligning two sequences of length m, n is described in Equation 2.3

$$\binom{n+m}{m} \frac{(m+n)!}{(m!)^2} \approx= \frac{2^{m+n}}{\sqrt{\pi \doteq m}} \tag{2.3}$$

Calculating scoring matrices for genomic sequence alignment is relatively simple since the mutation frequency for all the bases is equal. Highest positive score for exact matches, less positive score for substitution of characters. However, in biological sequence, "certain kinds of deletions or insertions are more likely than others" (Jurafsky & Manning, 2014) [40]. Basically not all gaps and substitutions have the same penalty. In DNA sequences, A and G are both purines whilst C and T are both pyrimidines, which makes transformations $A \leftrightarrow G$ and $C \leftrightarrow T$ complement transitions that occur more frequently. Other transformations are called transversions [24].

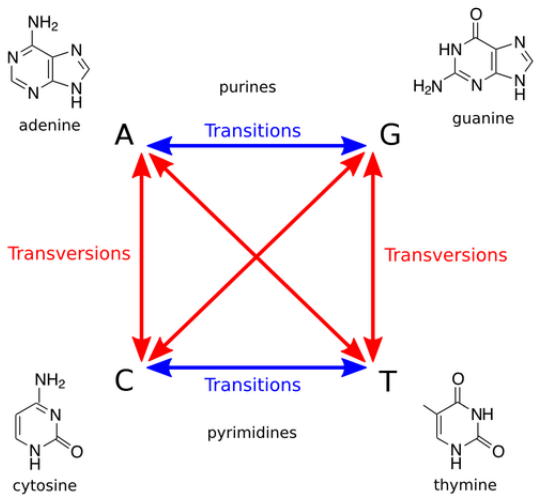


Figure 2.5: Transitions and Transversions [24]

Langmead states that "although there are twice as many possible transversions, because of the molecular mechanisms by which they are generated, transition mutations are generated at higher frequency than transversions." (Langmead & Pritt, 2016) [24]. He explains that the human transition to transversion ratio is approximately 2:1, hence transitions should be penalised less. The human substitution rate is one in 1000 bases whilst the indels rate is one in 3000 bases, explaining the fact that indels should be penalised more. These define the gaps penalty. The penalty matrix [24] is defined as: $transitions\,penalty \leqslant transversions\,penalty \leqslant gaps\,penalty$. Using the penalty matrix, a weighted minimum edit distance [40] can be computed as seen in Equation 2.4

$$D[i,j] = min \begin{cases} D[i-1,j] & + del[x(i)] \\ D[i,j-1] & + ins[y(j)] \\ D[i-1,j-1] & + sub[x(i),y(j)] \end{cases} \tag{2.4}$$

Setting up the penalty matrix or scoring matrix is split in either global alignment

or local alignment. Global alignment gives the user the ability to set each penalty according to the biological problem at hand. In this case entire strings are to be matched. In local alignment, we are not finding distance between two strings but finding the most similar pair of substrings between x and y. This is different from the edit distance problem where the goal is to minimize cost of transforming one sequence to another. In this case, a scoring matrix is used instead of a penalty matrix, whereby matches are given positive scores. The goal is to find parts which are similar and whose score is maximum across all possible substrings, hence pop out in the matrix. Eventually, once the maximum score in the matrix is found, a traceback to 0 is performed in order to figure out the most similar substrings.

Needleman-Wunsch (NW) [41] and Smith-Waterman (SW) [10] algorithms are the main algorithms used for sequence alignment. Both algorithms apply the dynamic programming based approach. The main difference is that Needleman-Wunsch [41] is based on global alignment, where the entire strings need to be matched. On the other hand, Smith-Waterman [10] is based on local alignment where the problem is to identify highly similar substrings. Global alignment can result in lots of gaps if the sizes of the query and the reference are dissimilar.

Needleman-Wunsch Algorithm (Global Alignment)

The Needleman-Wunsch algorithm performs global alignment. Given two sequences of length n and m respectively, a matrix having $n+1$ columns and $m+1$ rows is created. A simple basic scoring schema is defined in [25]. If two base pairs at ith and jth position are matching, a score of 1 ($S(i,j) = 1$) is assigned, whilst if the two base pairs at ith and jth position are mismatching, a score of -1 ($S(i,j) = -1$) is assigned. In case of a gap a score of -1 is assigned. The highest score identifies an optimal alignment. As explained in vlab.amrita.edu (2012) [25], the NW algorithm "consists of three steps:

1. Initialization of the matrix with the possible scores

2. Matrix filling with maximum scores

3. Trace back the residues for appropriate alignment"

Time and space complexity are defined as $O(nm)$. The time complexity for n sequences of average length L is $O(L^n)$.

Smith-Waterman Algorithm (Local Alignment)

The Smith-Waterman (SW) algorithm builds upon the Needleman-Wunsch algorithm. It determines whether an optimal alignment can be found for all alignments irrespective of length, location and sequence. This algorithm performs local sequence alignment, meaning that segments of all possible lengths are compared in order to optimize the similarity measure. Scores are assigned for each character-to-character comparison in the following procedure: highest positive score for exact matches, less positive score for mismatches, whilst negative score for insertions/deletions. Scores are added together and the highest scoring alignment represents the optimal local alignment. This algorithm is defined formally in [66] as shown in Equation 2.5

$$v(i,j) = max \begin{cases} 0 & \\ v(i-1,j-1) & + cost(S_1(i), S_2(j)) \\ v(i-1,j) & + cost(S_1(i), -) \\ v(i,j-1) & + cost(-, S_2(j)) \end{cases} \tag{2.5}$$

The matrix has $(m+1) * (n+1)$ entries (where the value of each cell is $v(i,j)$) and can be filled either row-wise or column-wise and each entry can be filled in constant time. Thus time complexity is $O(mn)$.

In practice, dynamic programming is slow hence it needs to be combined with an index for approximate matches. The idea is taking the best of both worlds, that is, whilst dynamic programming handles mismatches and gaps with the drawback

of being slow, an index is very fast and good at narrowing search space but does not handle mismatches and gaps.

2.3.11 Burrows-Wheeler Aligner's Smith-Waterman (BWA-SW) Alignment

BWA-SW [52] algorithm constructs an FM-Index of the query sequence Q and the reference sequence R. A prefix Directed Acyclic Word Graph (DAWG) is built using query sequence Q. A Prefix Trie (PT) is built using the reference sequence R. The prefix trie is a tree representation of sequence R. A node in prefix trie represents a string as shown in Figure 2.6. The unique string obtained is a substring of the sequence R. Each node of the tree is represented using suffix array interval. Traversing nodes generates strings that are lexicographically sorted. PT(R) and Prefix DAWG(Q) are initially computed. The best score between the sequences Q and R is computed using SW algorithm. To optimize the computations, the reverse postorder traversal scheme on both is also adopted. The dynamic programming mechanism in the BWA-SW algorithm enables identifying the seed matches of the genomic sequences.

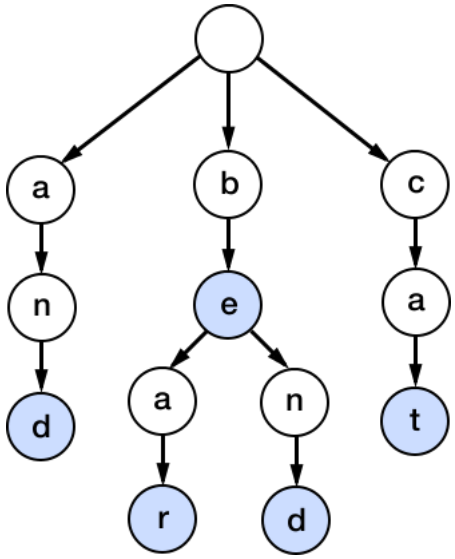


Figure 2.6: Prefix Trie for the English words *and, be, bear, bend, cat.* [42]

2.4 Related Work

The major problem faced by bioinformaticians when using huge data sets is that they need machines having high computational power requirements which are expensive to use.

Stephens et al. [5] discuss key technologies needed in relation to big data genomics based on four analyzed components:

- Acquisition - sustaining growth is possible by continued advances in NGS which improve throughput and accuracy whilst reducing costs. Ideally, sequencing time must be reduced to real time to enable rapid diagnosis of acute infections.
- Storage - Efficient compression and indexing systems are critical for highly accessible data. Personal genomes can also be presented as compact graph.
- Distribution - Continue investing in cloud computing in order to minimize data movement. However, data should be available through APIs, keeping in mind issues such as authentication and encryption.
- Analysis - Large scale machine learning is needed in order to be able to analyze how DNA mutations relate to disease, development and behaviour. For this reason, continuous development and improvement on highly scalable systems such as Apache Hadoop is expected. However, it is ideal to start integrating data science concepts to undergraduates leading to better bioinformaticians, such that new highly optimized algorithms are implemented.

Kashyap et al. (2014) [43] list five main types of data that is heavily used in bioinformatics research: "i) gene expression data, ii) DNA, RNA, and protein sequence data, iii) Protein-Protein Interaction (PPI) data, iv) pathway data, and v) Gene Ontology (GO)." Projects for genome population sequencing research are increasing at a larger scale. Case in point is the Genome 100K project[2] which evolved from the previously Genome 10K project[3]. Genome 100K aims to sequence

[2] `http://www.genomicsengland.co.uk/`
[3] `https://genome10k.soe.ucsc.edu/`

the DNA of 100,000 individuals from the UK to better understand the mechanisms of many diseases. The human genetic diversity has been tackled previously in projects such as the 1000 Genomes Project [44]. Moroeover, this project describes thoroughly genetic human diseases [45]. The 1000 Genomes Project led to a much more accurate information on several patterns of human DNA variation. The data produced in the 1000 Genomes Project is estimated in 100TB, whilst data in the 100K Genomes Project will be 100 times bigger in size [46]. Similar projects, albeit in a smaller scale, are also being put forward locally in Malta. Comparisons of genome sequences produce matching alignments and similarity scores. These similarity scores represent the similarities/dissimilarities between these biological sequences. The matching alignments and similarity scores are then used for secondary structure predictions. With the advances in sequencing technologies, millions of sequences with greater read lengths, that is, $>$ 1000 base pairs are now being generated. Based on genomic data, the sequencing tools can be basically classified into two categories [8]:

1. Short read aligners: used to align genomic sequences whose read length is between 32 base pairs and 200 base pairs, for example, BWA-backtrack [9] and Bowtie2 [47].

2. Long read aligners: used to align genomic sequences whose length is greater than 1000 base pairs, for example, Burrows-Wheeler Aligner Smith-Waterman (BWA-SW) Alignment [52].

Other examples of short read aligners include BLAST [49], BLAT [50] and SOAP [51]. For long read alignment, there are also Cushaw2 [53] and BWA-MEM [54].

Abuin et al. [4] describe how BWA software consists of three algorithms: BWA-backtrack [9], BWA-SW [52] and BWA-MEM [54]. BWA-backtrack [9] is the first algorithm produced from the three and is based on a backtracing search procedure using BWT [55]. It works best on SRA with a large reference sequence such as the human genome by allowing mismatches and gaps. BWA-backtrack was orignally

designed to cater for short sequence reads up to 100 base pairs. BWA-SW [52] extends on BWA-backtrack [9] by using only a few Gigabytes of memory to align long sequences up to one million base pairs against the human genome. This was introduced to handle longer reads since BWA-backtrack was specifcally built to handle short queries. In this case, seeds are found by dynamic programming (using SA Algorithm [10]) between two FM-indices. Mismatches and gaps are allowed into the established seeds. Seeding is finding exact matches of part of the read with part of the target. The seeds serve as a base in finding possible locations for the reads. A seed is extended when the same seed has few occurrences in the reference sequence. The latest algorithm in BWA, BWA-MEM [54] is also designed for long reads. Local and end-to-end alignments are automatically choosen. BWA authors (bio-bwa.sourceforge.net, 2010) specify that "BWA-MEM algorithm is recommended for high-quality queries as it is faster and more accurate. BWA-MEM algorithm also has better performance than BWA-backtrack algorithm for 70-100 base pair Illumina reads[4]"

Aligners can also be aggregated as is the case of Crossbow [22]. This launches distributed copies of Bowtie2 [47] on Apache Hadoop. After generating the aligned reads, MapReduce is used in order to sort and aggregate the alignments. Lin et al. [59] claim that in the benchmark set on the Amazon EC2 cloud, a human sample consisting of 2.7 billion reads was aligned on Crossbow in less than three hours using a 320-CPU cluster, for a total cost of $85.

Lee et al. [60] explain that the two paradigms used for aligning reads with the reference genome are either based on the Burrows-Wheeler Transformation with FM-Indexing (BWT-FM) or else using seed-and-extend. Specifically they mention that, "mapping Complete Genomics (CG) reads with BWT-FM is a harder problem since the expected gaps in CG reads show themselves as indels in an alignment, which increases the runtime of BWT-FM search exponentially" (Lee et al., 2015) [60]. Since BWT-FM aligners perform gapped alignment per read, this makes them

[4] `http://bio-bwa.sourceforge.net`

impractical to use for CG reads as in this case, both ends contain gaps. A gapped alignment is achieved by anchoring one end of the sequence read with a gap-free alignment. Lee et al. [60] point out that when having high error rate, notably errors related to indels, seed-and-extend techniques scale better. Seed-and-extend methods are recommended, despite the fact that BWT-FM methods are betterin aligning consecutive reads having minimal sequence errors. In point of fact, they design a new read mapper, sirFAST, using a hash based seed-and-extend algorithm. This seed-and-extend mapper has two main steps. First, seeds are selected from a set of short subsequences of a read. The locations of such seeds in the genome is queried using a hash table. The similarity score between the input read and the reference genome, is then calculated in a verfication step performed by the mapper. This verification step score is obtained by calculating the alignment of the full read against chunks of the reference genome at the previously extracted seed locations. As explained by the authors, "this verification is more computationally expensive than verifying conventional reads of consecutive bases since the similarity score of a CG read to a reference, is calculated as the sum of the minimum similarity scores of all read sections." (Lee et al., 2015) [60]. To compensate for this, Lee et al. [60] use a combined seeds technique allowing for a remarkable reduction in the search space of CG. This optimization is based on the fact that any two of the 10 base pairs read sections are merged together as a single merged seed. Hamming distance [61] is preferred over Levenshtein [62] or SW [10] since in the last two algorithms, run time improvements are limited due to the large amount of expected gaps. sirFAST was able to map all simulated reads. Also, sirFAST obtained higher percentage mapping than other CG mappers when using real sequence reads data and mapping these reads to the reference human genome assembly (GRCh37) using a maximum edit distance of two.

Gonzalez-Dominguez, Liu & Schmidt [63] present a novel approach for SRA by taking advantage of a distributed shared memory programming model based

on the UPC++ language [5]. UPC++ uses Partitioned Global Address Space and provides a private memory space per process for local computation. It has been used in other parallel sequence alignment research [64]. Gonzalez-Dominguez et al. [63] use CUSHAW3 [65] as their base short-read aligner because of its high alignment quality. According to authors, "this approach provides the same high quality mappings as the original CUSHAW3, but reduces drastically the runtime when executing on several nodes." (Gonzalez-Dominguez et al., 2016) [63]

2.4.1 Distributed Alignment Techniques

Venkatachalam [66] presents an approach whereby filling up the matrix entries can be computed in parallel. His idea revolves around the fact that all the elements of the anti-diagonal depend on the previous anti-diagonal but are independent of each other, hence the parallelism. An anti-diagonal is filled on every iteration, instead of filling a row or column. Having $n * m$ matrix, the longest diagonals are of length m and there are $n - m + 1$ such diagonals. Hence using m threads will achieve best throughput, however leading to $(m - 1)$ stalls as threads will be idle in computation.

The author proposes some engineering optimisations in CUDA. NVIDIA (developer.nvidia.com/ cuda-zone, 2017) state that "CUDA is a parallel computing platform and programming model invented by NVIDIA based on graphical processing units"[6]. Sequence comparisons are parallelized and every process has an equal length of sequence to compute. Multiple rows per thread should be assigned to minimize overhead, since every thread computes one value and waits for synchronization. Moreover, loads and computations can be pipelined, that is, after the first string is loaded in memory and kernel is processing it, other strings can be loaded into memory. This results in performance optimization since it does not drop by idle GPUs.

Another parallelization paradigm for computing sequence alignment is by using

[5] `https://bitbucket.org/berkeleylab/upcxx/wiki/Home`
[6] `https://developer.nvidia.com/cuda-zone`

MapReduce and cloud computing. CloudBurst [67] uses MapReduce for SRA of subsequent genomic sequencing reads to a reference genome whilst allowing for the end-user to specify dynamically the number of mismatches. CloudBurst genomic alignment scales linearly with the number of reads using the underyling architecture of HDFS. Schatz (2009) explains that CloudBurst "is based on seed-and-extend mechanism that indexes the non-overlapping *k-mers* in the reads as seed. The seed size $s = \frac{r}{(m+1)}$ is computed from the minimum length of the reads (r) and the maximum number of mismatches (m)" [67]. The exact seed is extended using a dynamic programming algorithm in order to count the number of mismatches. These are used to report alignments having at most m mismatches. Nguyen et al. address the drawbacks in [67] by designing CloudAligner[68] which has a better performance and scalability. It accepts two input files; read file and reference file. Both files are changed into binary format and copied to HDFS. Upon executing, these files are split into smaller chunks and distributed along different maps. Each map aligns its chunk along the whole reference genome file. Matsunaga, Tsugawa & Fortes (2008) present CloudBLAST [69] - "a solution whereby query sequences are distributed and processed in parallel. Such an approach works fine and is fault tolerant for small files with many reads". The main drawback of CloudBLAST is that the performance drops perilously in case the size of input file containing the reads exceeds the local node's RAM. Moreover, it lacks load balancing causing non-optimal use of the cluster. In case, there are smattering input query reads, several nodes are left idle since only few data nodes will execute.

Algur and Sakri [70, 71] present two similar approaches for sequence alignment using a Smith-Waterman Alignment parallelisation on the MapReduce. A customised MapReduce implementation based on Azure Cloud platform is developed. In [70], a work called *SW-PAMR - Smith-Waterman Alignment on the Parallel Azure Map Reduce* is presented, whilst in [71], an extension called *SW-BSPMR - Smith-Waterman Alignment on the Bulk synchronous Parallel Map Reduce*, is introduced. In both cases, the reference sequence is split into overlapping chunks

before sent to mapper. Eventually, the whole query sequence together with a chunk reference sequence is sent to the mapper. In each of the computing map workers, the query sequence is also divided into chunks and kept in the available local memory. Sequence alignment between each query sequence chunk and the reference sequence chunk is done using a parallelized SW algorithm, based on CUDA techniques as explained in [66]. However, the availability and the cost of such computing platforms on the public clouds is an issue. The multiple alignment positions and computed score are stored in the temporary cloud storage memory. In the reduce phase, all the non-overlapping and non-redundant alignment positions are aggregated. The application was deployed on the Azure cloud considering A3 VM instances. Each A3 VM instance consisted of four virtual computing cores, 7GB of RAM and 120GB of local hard drive space. Azure HDInsight was used as Hadoop implementation. The baker yeast genome database (Saccharomyces cerevisiae S288c)[7] was used for evaluation. Experiments were conducted using a constant reference genomic sequence and four query sequences of varied lengths. Authors claim a speedup of about 12.5 for 1K query sequence length and a speedup of 43.5 for 500K query sequence length, when compared to a similar system, *SW-Hadoop*. However, the authors never evaluated their system with bigger data such as the human reference genome and `FASTQ` human genome reads.

Sudha Sadasivam [29] proposes parallel dynamic programming based on three levels. First, all possible sequence permutations are generated and executed in parallel. Second and third stages parallelize alignment between independent pairs of sequences. All three stages, are performed using MapReduce programming paradigm whilst Needleman-Wunsch algorithm is used for pairwise alignment.

HBLAST [72] is a parallelised BLAST algorithm that uses a novel approach based on virtual partitioning in order to split both reference and input query sequences. In HBLAST, O'Driscoll et al. [72] build upon previous work - bCloudBLAST [73]. In [73], both input read sequences and database chunks are distributed amongst

[7]Saccharomyces genome database (SGD). (2015). `http://www.yeastgenome.org/`

multiple Hadoop clusters. However this solution lacks scalability since database sharding needs to be done manually according to the number of nodes, clusters and RAM. HBLAST [72] overcomes such drawbacks since it implements automatically adjustable database shard size according to the number of available nodes in the Hadoop cluster and the amount of input query sequences. This is made possible by using runtime virtual sharding of the database. O'Driscoll et al. (2015) [72] specify the following advantages over other approaches:

- Automated database partitioning
- Flexibility in case of limited RAM
- Improved load-balancing via virtual partitioning
- Improved scalability
- Little mapper launch overhead
- Deployment simplicity

The query is split according to the minimum and maximum number of database shards. The input file is split such that the number of virtual shards maps exactly with the number of map tasks. Each mapper in HBLAST has as input, a line from the split query file. The key is the byte offset of the file and the value is the query sequence data. Eventually, the required query sequence file and database partitions are retrieved from HDFS to the local file system. These are used by subsequent runs as a caching mechanism. The mapper output key is the query sequence ID. For each key, all file sequence IDs and their complementary identity scores, are mapped. Sorting is done according to identity score and the output produced by the reducer contains the queried sequence IDs sorted by identity score pairs. This is explained in Figure 2.7

O'Driscoll et al. (2015) [72] claim that HBLAST was deployed on a private cloud infrastructure having two 8-core Intel Xeon 2.1 GHz CPUs with hyper threading

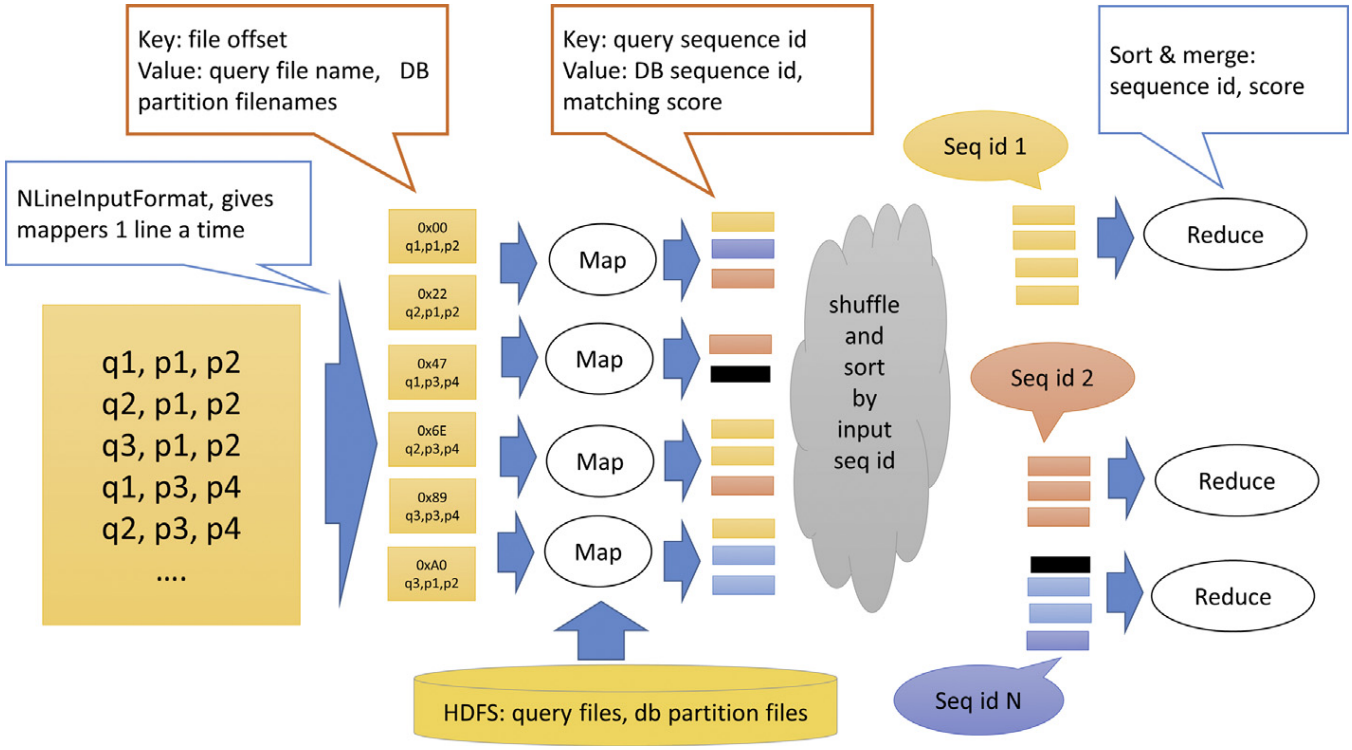


Figure 2.7: HBLAST MapReduce algorithm [72]

where each physical node contains 128 GB RAM. Processing time increases when using AWS Elastic Map Reduce (EMR). HBLAST is compared against CloudBLAST [69]. Using up to two million sequences, both have comparable performance, however as the database size is increased to five million sequences, authors claim that the performance of CloudBLAST degrades whilst HBLAST is 2.5 times faster.

Burrows-Wheeler Aligner's Smith-Waterman Alignment on Parallel MapReduce (BWASW-PMR)

Al-Absi and Kang [8] propose a cloud-based approach building upon BWA-SW algorithm (2.3.11) for long sequence alignment by using Hadoop MapReduce framework. Map and reduce phases are executed in parallel whilst utilizing all available cores in the data nodes in order to overcome the iteration drawback of Hadoop. SW algorithm is optimized as well. Reference sequence R is split into R' chunks with overlapping sections. The reference chunk and query Q are sent to map worker

nodes as input `<key, value>` pairs. Eventually Q is split into Q' chunks on each mapper worker node. Each Q' and P' are aligned using BWA-SW algorithm utilizing all cores of each node. BWA-SW is optimized by a parallelization technique to reduce execution time on each map worker. In the map phase of the BWASW-PMR cloud platform, alignment locations and corresponding scores between Q and chunk R' are computed. Similarly to BWA-SW, $PT(R')$ and $PrefixDAWG(Q')$ are computed. Also seed matches of Q' and R' are computed using dynamic programming mechanism.

For the reduce phase, the master node initializes reduce worker nodes simultaneously with map worker nodes. This parallelisation reduces total execution time. The reduce function in BWASW-PMR aggregates the alignment locations. The overlapping and redundant alignments are neglected. Parallelization of the reduce function is achieved by utilizing all computing cores available within worker nodes.

BWASW-PMR was developed using C++ and C#.Net and deployed on the Azure cloud platform. A3 VM instance consisting of four computing cores, 7GB of RAM, and 120GB of local hard drive space were used. Section of Homo sapiens chromosome 15, GRCh38.p2 Primary Assembly (NC 000015.10), was considered as the reference R. Query sequences considered were obtained from the influenza virus database[8]. Al-Absi & Kang (2015) claim an "average speed-up of 6.7 when compared to Bwasw-Cloud. Also, an average reduction of 30% in the map phase makespan is reported. Optimized SW algorithm, results in reducing the execution time by 91.8%" [8]. In this case, the authors again fail to test their system using bigger data such as the human reference genome and `FASTQ` human genome reads as happened in [70, 71]. In this case only human chromosome 15 was used as reference. This makes up only between 3% and 3.5% of the whole reference human genome.

[8] `http://www.ncbi.nlm.nih.gov/genomes/FLU/FLU.html`

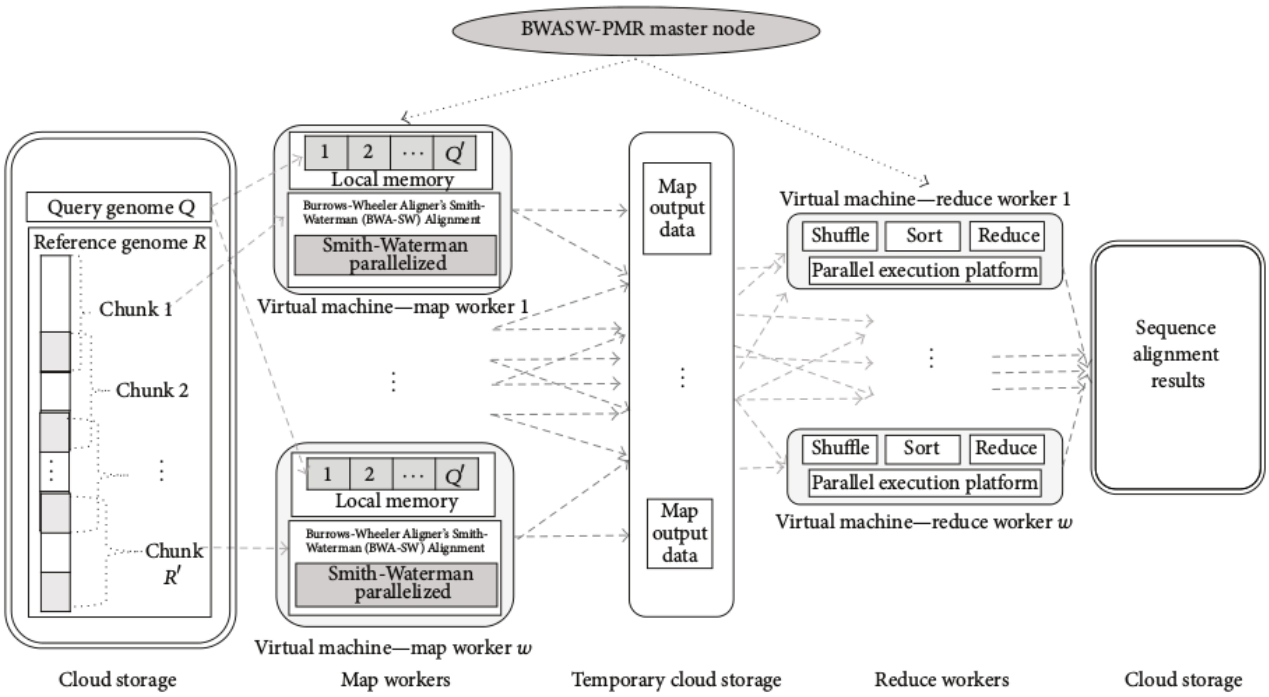


Figure 2.8: The BWASW-PMR architecture [8]

BigBWA

In their study, Abuin, Pichel, Pena, & Amigo, (2015) [4], introduce BigBWA[9], "a tool based Hadoop to increase the performance of BWA." They claim that the three main advantages of this system are:

1. Parellel alignment process resulting in reduced execution time
2. Fault tolerance capabilities based on the underlying Apache Hadoop technology
3. Original BWA was not modified

Abuin et al. explain that BigBWA consists of four steps:

1. Input `FASTQ` files are converted to a format compatible with Apache Hadoop
2. The converted files are copied to HDFS

[9] `https://github.com/citiususc/BigBWA`

3. Alignment is executed using BWA

4. Output is copied back from HDFS to the local filesystem

Abuin et al. (2015) [4] state that "this system was tested using data from the 1000 Genomes Project [44] running on a five-node AWS cluster with 16 cores per node (Intel Xeon E5-2670 at 2.5 GHz CPUs), using Apache Hadoop 2.6.0. BigBWA outperforms similar tools like pBWA [56] and SEAL [57]." SEAL is based on Pydoop [58], which implements the MapReduce programming paradigm using the Python programming language. However, according to Abuin et al. , this is considered as an overhead when compared to an implementation using JNI. pBWA uses Message Passing Interface (MPI) to parallelize BWA, however this lacks the fault tolerance capabilities offered by the underlying architecture of Apache Hadoop. Abuin et al. (2015) [4] claim that "using small number of cores, BWA is slightly better than BigBWA, however when using 16 cores, BigBWA always performs best." A main drawback of BigBWA is that its codebase is not maintained and hence became obsolete. Moreover, it does not offer the possibility to reassemble human genome of the sample.

This chapter presents various ideas in tackling genome alignment problem. However, the related work described, do not offer the possibility to reassemble the human genome of a biological sample from the alignment output and compare it with the reference human genome. Moreover, the output just provides genome alignment without generating any insights about the results which can be used to analyze genomic variations. Further to this, not all software presented in this chapter is regularly maintained. We overcome these bottlenecks in our approaches presented in the next chapter.

3. Methodology

In this chapter, we outline the design and implementation of our approaches towards distribution of the sequence alignment process. Our three approaches are:

1. `SW-Optimization`: The first original approach is based on optimizing a native implementation of Smith-Waterman algorithm for local alignment of genomic sequences. This approach did not produce desired results since our implementation suffered from out of memory errors as explained in Section 3.4, hence, our focus shifted to the remaining two approaches.

2. `MR-BWA`: This presents an approach in distributing BWA using MapReduce. BWA [9] is a single node industry standard software used for genomic reads alignment.

3. `MR-BWT-FM`: In this approach, we re-implemented BWT (the main algorithm behind BWA) in a distributed manner while also implementing low level optimizations on suffix array and BWT creation. These are then used to create a custom FM-Index. Finally, this custom FM-Index is used for distributed genome sequence alignment based on MapReduce.

All three approaches have been implemented using Python v2.7.6[1]. Distribution for `MR-BWA` and `MR-BWT-FM` use the MapReduce programming paradigm defined

[1] `https://www.python.org/download/releases/2.7.6/`

in Section 2.2. Apache Hadoop v2.7.1[2] has been chosen as the open-source implementation of MapReduce. Specifically, the Hadoop streaming API v2.6.5[3] has been used. Though Apache Hadoop framework is implemented in Java, in our case Python was preferred as it is more expressive. Moreover, Python is commonly used in bioinformatics, given the OS and technology support it has. In our case, we used Ubuntu Linux v16.04 LTS for compatibility reasons. For streaming applications, Python with Apache Hadoop is used to process and analyze large data sets.

3.1 Input

The input format for our three approaches is the industry standard `FASTQ`. `FASTQ` text files are the output files generated by NGS technologies such as Illumina. Cock et al. (2010) [74] explain in detail the structure of a `FASTQ` text file. Each read consists of four lines. The first line is a read identifier starting with an `@`. The second represents the actual DNA read, the third line is another identifier, similar to line one, but starting with a `+` (in certain cases it consists only of a `+`), whilst the forth contains a calculated Phred quality score symbol for each base pair in the read. As explained in [75], this Phred quality score is based on the ASCII character code[4]. This score is defined mathematically as: $Q = -10 \log_{10} p$, where p is the estimated probability of a base call being incorrect. High Q values correspond to more confidence in base being correct. Moreover, whilst analyzing the read files one can notice that end of reads have low quality values such as two.

3.2 Output

Output to our approaches is a Tab Separated Values (TSV) file specifically designed by us, containing four columns:

[2] `https://hadoop.apache.org/docs/r2.7.1/`
[3] `https://hadoop.apache.org/docs/r1.2.1/streaming.html`
[4] `http://www.ascii-code.com/`

1. 0-based human reference genome base pair position
2. Human reference genome base pair found at position listed in first column
3. Chromosome name related to the position listed in first column
4. Combined counts of base pairs at position listed in first column, related to the human sample that has been aligned. This is represented via a csv list of six integers representing counts for `A,C,G,T,D,N` respectively. `A,C,G,T` represent the DNA base pairs, `D` stands for deletion in person's genome whilst `N` stands for a no-call in query read. A detailed example is explained below.

Example 3.2.1. Output row

```
38952   A   chrX   50,0,0,5,0,0
```

means that 55 input reads had 1 base pair aligned at position 38952 of the human reference genome. Out of these 55 base pairs; 50 were `A` whilst five were `T`. The actual base pair at position 38952 of the human reference genome is `A`, which means that in this case we can safely assume that the particular human in question does not have any variations for this specific position since 91% of the reads matched the human reference genome.

3.2.1 Output Analysis

The above format is generic, in that it can be easily analyzed just by parsing tab values or opened in a spreadsheet. We designed such output in order to present a simple way to reassemble the human genome of the sample and compare it with the reference human genome. Such feature is not available in related work such as `BigBWA` [4]. A script has been implemented to parse the proprietary output generated by our application and generates insights and charts about the results. This is further explained in Appendix C. The main goal of the script is to extract the read base pair having highest count per line and compare it with the reference

genome base pair. If the base pair is different, this script takes into consideration zygosity. Since humans inherit a set of chromosomes from the father and another from the mother, hence homologous chromosomes can have same or different alleles. In case of having same alleles then individual is homozygous for that particular gene. If alleles are different, then individual is heterozygous for that gene. This means that although reference base pair and read base pair are different at a particular position, this does not necessarily mean there is a variation. In our case, keeping in mind that reads base pairs at a particular position are ordered in descending order of count, as soon as a variation is observed, we check if the next base pair in line is within a predefined threshold of 10% from the previous one and also has at least 30% of the total number of reads aligned at this particular position.

Example 3.2.2. If for a particular position, the base pair of the reference genome is `A` and reads are distributed as 45% `A`, 50%`C`, 5% `T` at that position, this is still considered as a match since the next read in line, that is, `A` has only 5% less reads than `C`. Moreover `A` has more then 30% of the total count of reads aligned at this position. In this case the individual is heterozygous.

Having the base pair count per matches, variations and indels, an alignment operation chart is created whereby distribution can be analyzed. Variations and/or indels per chromosome can also be extracted. Example of generated charts which have been used for correctness evaluation are presented in Section 4.4.

The input files are uploaded on cloud storage and the generated output files are also stored automatically on cloud storage. Eventually, these output TSV files are downloaded on local machine for analysis.

3.3 MapReduce

As explained in Section 2.2, the MapReduce programming paradigm, is based on two main functions: `map` and `reduce`. In our case an intermediate optional

function, `combiner`, has also been used. The combiner step is an optimization which aggregates mapper output and minimizes the time taken for data transfer between mapper and reducer.

3.3.1 Mapper

In our case the `map` function does the majority of the work. Each mapper is divided into two stages: pre-processing and post-processing. The pre-processing stage parses the input reads files and align these samples with the human reference genome. The post-processing part analyzes the alignment result in order to extract the columns described in Example 3.2. The mapper output is in the following format:

- `key`: reference genome position
- `value`: a tuple containing reference genome base pair, reference genome chromosome name and the combined counts of `A,C,G,T,D,N` for all query reads that aligned for that particular reference genome position in a specific map task. These 3 elements in the tuple are separated by semi-colon (`;`)

More formally the mapper output is:

```
<reference_genome_position,
(reference_genome_char; reference_genome_chromosome_name; combined counts of ACGTDN)>
```

The reference genome position is used as key such that all alignments matching a specific reference genome position are sent to the same reducer. The three approaches differ in the `map` function implementation, however all produce the same mapper output.

3.3.2 Combiner

In our case, an optional combiner was also used in order to aggregate mapper output having the same key. For instance, if we have multiple output related to the same reference genome position coming from the same mapper, the combiner aggregates

the counts of `A,C,G,T,D,N` and only one `<key, value>` pair for the same reference genome position is outputted by the combiner. This minimizes the time taken for data transfer between mapper and reducer. Moroever, the reducer receives less data which as a consequence increases reducer performance and reduces network traffic. For every mapper there is at most one combiner, however MapReduce can decide not to run the combiner, e.g., in case a mapper is slow. For this reason, the combiner output is identical to mapper output.

3.3.3 Reducer

The `reduce` function aggregates counts for a specific reference genome position outputted from different mappers or combiners. This ensures that a specific reference genome position is outputted only once by the reducer. The reducer output is explained earlier in Example 3.2. The three approaches all have the same `reduce` function, since mapper and combiner output is the same for all of them.

3.3.4 Data Distribution

Data at hand (either reference genome or reads) need to be distributed and put onto HDFS. There are two main options for data distribution:

1. Chunk the reference genome and distribute chunks on HDFS. Reference genome chunks must have overlaps the same size as reads length in order to avoid missing any alignment. In this case all reads are sent on each mapper. This approach has been used in existing work as explained in Section 2.4.1. However, in this case, extracting the reference genome alignment position would be complex since only the alignment position with respect to the split reference genome part is known. This means that alignment with each chunk, needs to be aware of all other alignments to extract the global reference genome alignment position. This approach is shown in Figure 3.1

2. Distribute reads on HDFS whilst storing the whole reference genome on each

mapper. This approach allows for independent reads alignment that can be aggregated in the reduce stage. In this case, extracting the reference genome position is not a complex task and is done as explained in Section 3.2. The major drawback of this approach is that the reference genome index needs to be stored on each mapper, however this is done once offline. This approach is shown in Figure 3.2.

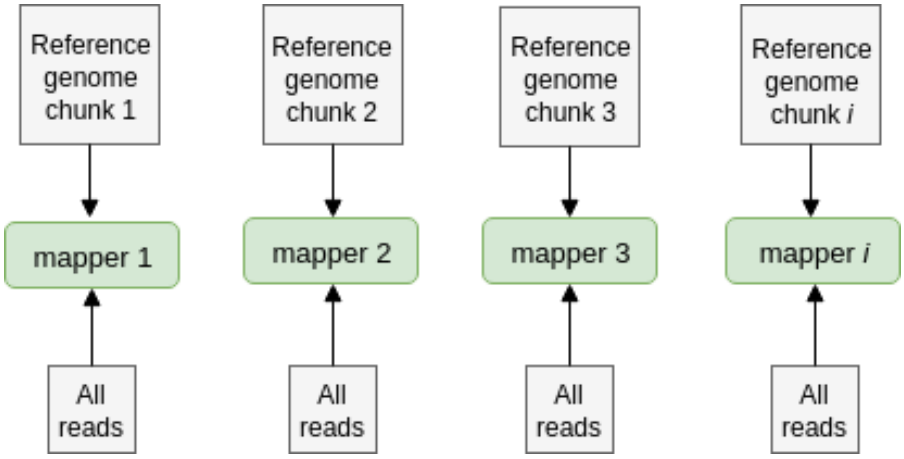


Figure 3.1: Data distribution using split reference genome. In this case, each mapper contains a chunk of the reference genome and all reads.

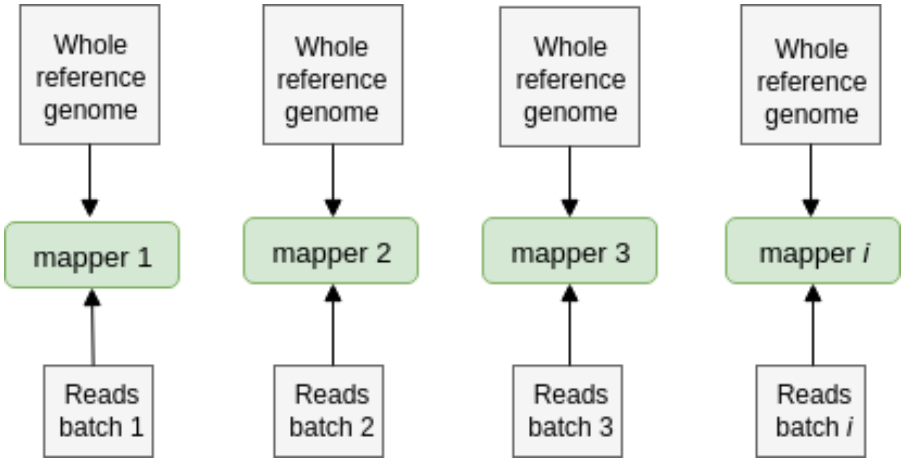


Figure 3.2: Data distribution using split reads. In this case, each mapper contains the whole reference genome and subset of the reads.

3.3.5 System Overview

An overview of the system architecture is shown in Figure 3.3. A complete MapReduce example using sample genomic data and how it moves from `mapper` to `combiner` and finally to `reducer` is shown in Figure 3.4. In this example one can observe how multiple mapper output for reference genome position `111` is first combined and then reduced to a single `<key, value>` pair. Read base pairs counts for reference genome position `111` are aggregated in the final reduce stage. In both figures, we note that combiners execution is optional and there is less reducers than mappers. If the number of reducers is not set explicitly, then it depends on the Hadoop cluster configuration.

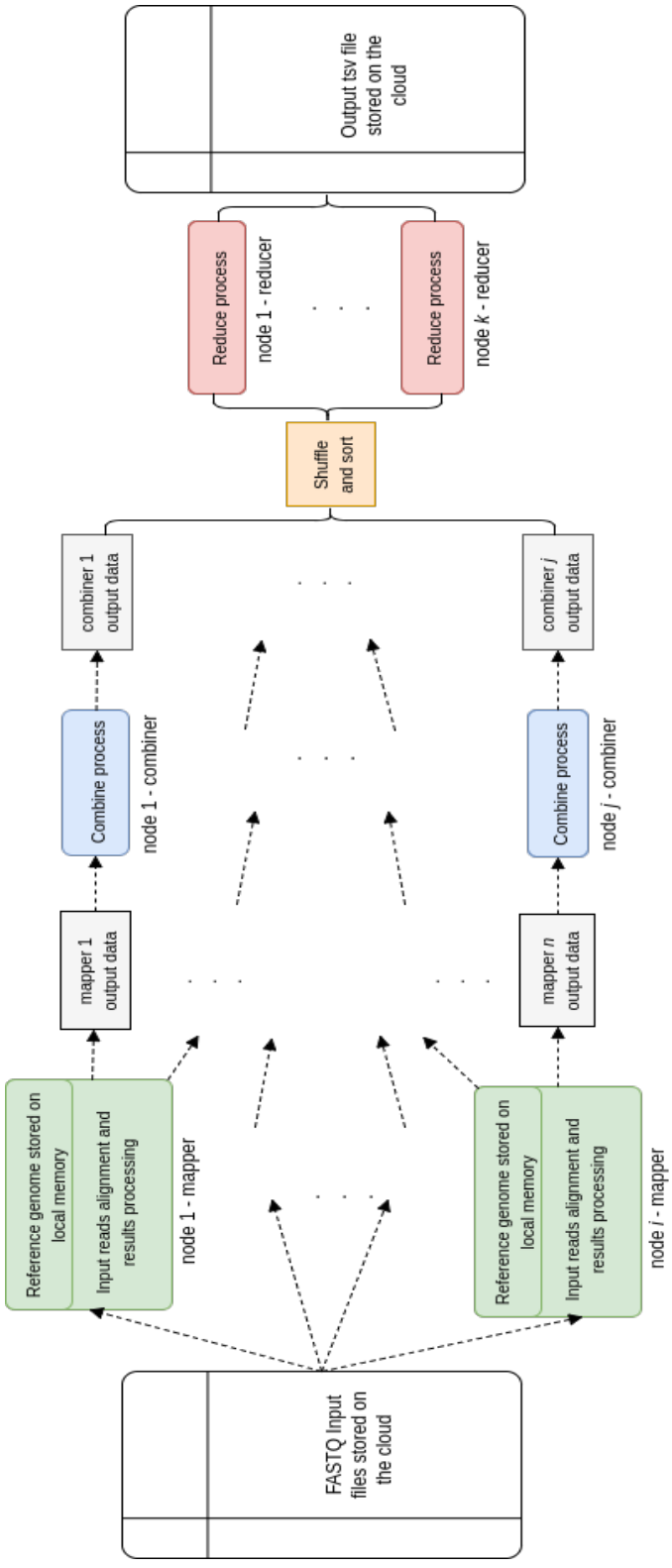


Figure 3.3: Overall system architecture. Each mapper aligns subset of reads with the whole reference genome. Mapper output is aggregated by combiners and reducers. Output TSV file is stored on HDFS.

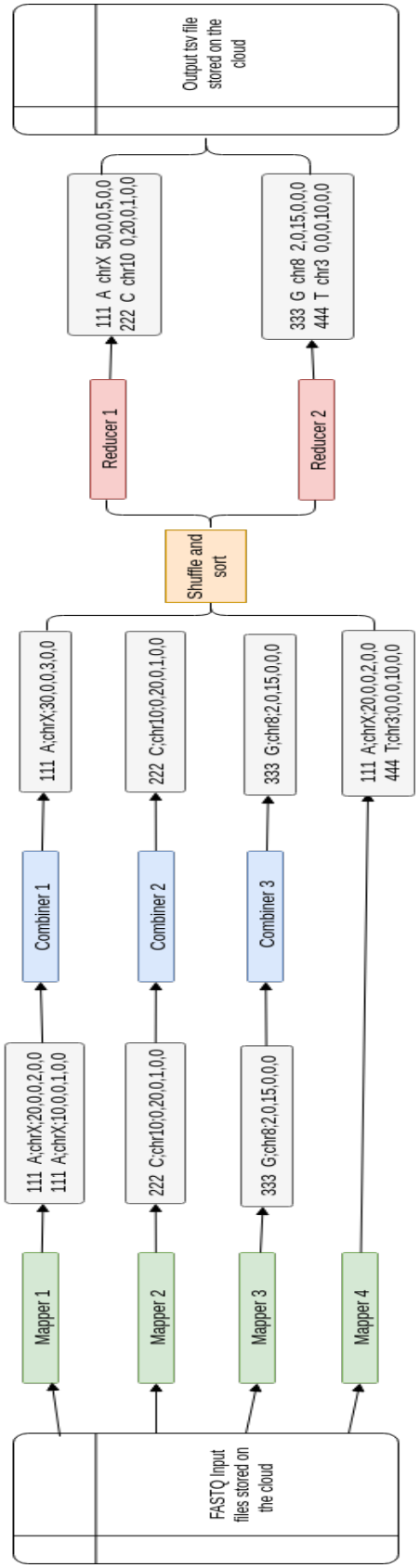


Figure 3.4: Genomic data MapReduce example. We can observe how multiple mapper output for reference genome position `111` is first combined and then reduced to a single `<key, value>`. Moreover, we note how in case of Mapper 4, the combiner was not executed and data was forwarded straight to reducer.

3.4 SW-Optimization

Our first approach is based on an optimization of the Smith-Waterman algorithm described in Section 2.3.10. In this case, a native implementation of Smith-Waterman algorithm for local alignment of genomic sequences was implemented. The program takes two strings as input and creates the scoring matrix. In our case the two strings were represented by the reference genome and each read coming from the input `FASTQ` file. The $(m+1)*(n+1)$ scoring matrix represents trial alignments of the read with the reference genome, where m and n are the respective length of the reference genome and each read. The extra row and column are needed for gaps (-). Each cell score is calculated using a function called `calc_score(matrix, x, y, seq1, seq2)` which calculates the score for a given x, y position in the scoring matrix and returns the best score defined as `max(0, diag_score, up_score, left_score)`. In case of having multiple maximum score, standard implementations take the first maximum score in order to reduce computations over the whole matrix. This was improved in our case by choosing the best alignment based on better base quality and better surrounding context, that is, the neighbouring reads also have high score. In our case the scores used are:

- MATCH_SCORE = 2
- MISMATCH_SCORE = -1
- GAP_SCORE = -1

The maximum score position in the scoring index is noted and is used to traceback in order to find the optimal path through the scoring matrix that corresponds to the optimal local sequence alignment. Each traceback move corresponds to a match, mismatch, or gap in one or both of the sequences being aligned. Moves are determined by the score of three adjacent cells: upper, left, and diagonal upper-left where: diagonal upper-left means match/mismatch, upper means there is a gap in first sequence whilst left refers to a gap present in the second sequence. The

traceback function returns the two aligned sequences, hence the reference genome start position has to be extracted manually using the maximum score position and the length of the aligned sequence as defined in Equation 3.1:

$$\begin{aligned} ref\_genome\_start\_position = max\_score\_pos.y- \\ query\_alignment.length+ \\ ref\_genome\_alignment.gaps \end{aligned} \tag{3.1}$$

Here we are simulating the traceback functionality by subtracting the number of cells from the maximum score position in order to obtain the start of the alignment. Any gaps introduced in the reference genome need to be added since obviously these are not actually in the original human reference genome. Once we have the human reference genome base pair alignment start position, all properties mentioned in Example 3.2 can be extracted.

This approach gives accurate results however it would need a powerful computer having over 1TB of RAM to align reads with reference genome. Creating the scoring matrix of $3 * 10^9$ (human reference genome size) $* 60$ (read size) $* 4$ (integer size) for each read would need approximately 720GB RAM, which is impractical in our case. For this reason, we scrapped this approach and focused on implementing distributed sequence alignment of genomic data based on the MapReduce programming paradigm. As discussed in Section 3.3.4, splitting the reference genome and creating a scoring matrix per chunk, requires that the split reference genome must have overlaps the same size as reads length, which results in increasing the total size used by each scoring matrix per chunk. Hence, we insisted on the novel approach of not splitting the reference genome and distribute via maps only the input read files. This is presented in the next approach which is based on parallelizing the BWA as explained in Section 2.3.11.

3.5 MR-BWA

The majority of the BWA distributed approaches mentioned in Section 2.4.1 chunk the reference genome and use the overlapping approach. This works fine where the final output is just the alignment file without doing any analysis on the reference genome positions. In our case, we opted for a novel approach whereby the reference genome is not split, in order to be able to identify the reference genome alignment positions whilst improving on the speed used in the existing approaches. Assuming that a Hadoop cluster is up and running with BWA pre-installed[5], this approach consists of the following steps (installation details are provided in Appendix A):

- Index and distribute the reference genome on each worker node
- Convert the input `FASTQ` files to a Hadoop compatible format (This is further explained in Section 3.5.2)
- Copy the preprocessed input on HDFS
- Run parallel BWA (Map, Combine, Reduce tasks)
- Copy output TSV file to master node from HDFS

3.5.1 BWA Python Implementation

The first step was to create python libraries that interact easily with BWA software since BWA is written in C. We aimed at having an application that is version-agnostic regarding BWA, hence assuring that our artifact is compatible with any BWA version without doing any code changes. For this reason, a library called `pyBWA` has been implemented. This library is made up of two main scripts: `bwa.py` and `seqio.py`. `seqio.py` contains utility functions checking type and validity of input files. The integration is found in `bwa.py`. Since BWA has different commands[6], a parent class has been implemented exposing common functions for BWA commands.

[5] `https://sourceforge.net/projects/bio-bwa/files/`
[6] `http://bio-bwa.sourceforge.net/bwa.shtml`

This holds two main properties that contain the BWA installation path and the BWA command to execute. Any class extending this parent needs to specify these two properties. The parent class also exposes a key-value hashtable with any extra arguments that subclasses may specify depending on the command to execute e.g. $-t = 4$ referring to using four threads during the BWA alignment process. These are parsed and added to the final BWA command to execute. The parent class uses `seqio.py` to ensure that an index exists prior to running an alignment. The main function in this class is called `run()`. This function, first checks that BWA is installed and is found in the correct specified path. Eventually, a python subprocess is spawned calling BWA with all the specified options. In case of success, the BWA alignment output is validated and written to file. If the BWA process failed, the program halts immediately whilst logging the error encountered.

Two subclasses have been implemented representing the `index` and `mem` commands in the BWA software. `BWAIndex` subclass only requires an input `fasta` file to index. Prior to creating the index, it checks whether an index for the specified `fasta` file already exists. `BWAMem` subclass requires two arguments; the indexed reference genome and the input `FASTQ` file to map reads from. Class diagrams for these two classes are shown in Figures 3.5 and 3.6. `BWAMem` subclass output is validated by checking that the count of reads returned matches the count of reads in the original input `FASTQ` file. BWA-MEM algorithm was chosen over other algorithms in the BWA software package since it is recommended by the BWA authors themselves (bio-bwa.sourceforge.net, 2010) "for high-quality queries as it is faster and more accurate."[7]

[7] `http://bio-bwa.sourceforge.net`

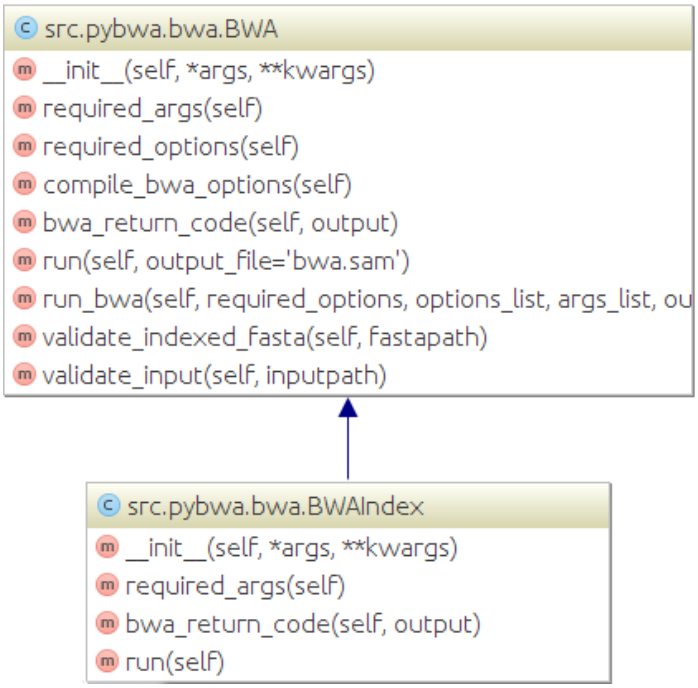


Figure 3.5: `BWAIndex` class diagram

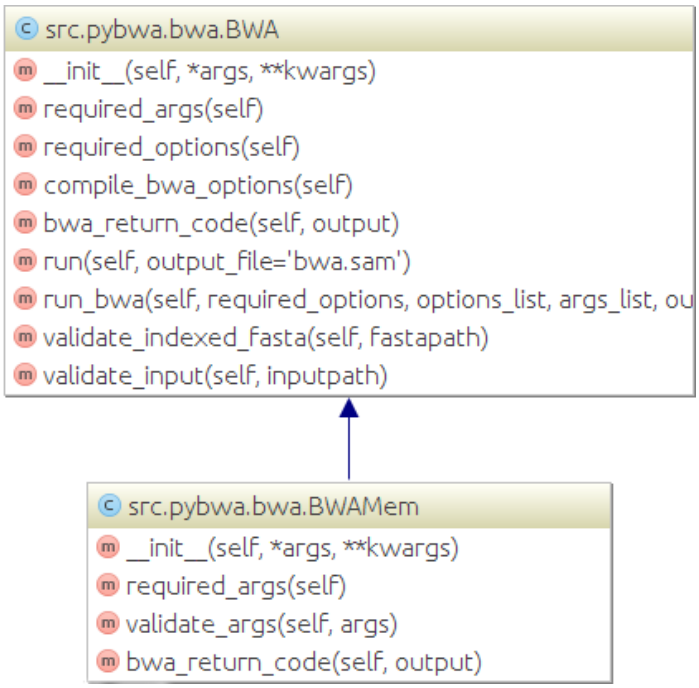


Figure 3.6: `BWAMem` class diagram

3.5.2 BWA Distributed Alignment

As explained in Chapter 3, `FASTQ` files have four lines for each read. On the other hand, Hadoop streaming API, by default, interprets each line of the input file as an individual entry for a mapper task. Therefore, it is necessary to preprocess the input files in such a way that one read (four lines in the original format) corresponds to one line in the Hadoop input file. A Python script `fq_to_mrfastq.py` has been created to adapt the `FASTQ` files to Hadoop format. This is done by joining every four lines into one line separated by `<sep>`. These custom preprocessed `FASTQ` files are uploaded to HDFS to act as input to mappers. Eventually each line in the custom preprocessed file is split into four entries again in mapper, in order to re-assemble the original `FASTQ` file before passing it to BWA.

Each mapper is divided into pre-processing and post-processing as shown in Figure 3.7. The pre-processing part starts by checking that a valid index file exists on the associated data node. If not, the mapper halts immediately. Eventually, the mapper parses the preprocessed input streamed from HDFS, such that the mapper has data which is in correct `FASTQ` format. The `FASTQ` input data is passed to `BWAMem` class, together with the human reference genome. Since BWA writes its output to disk, each mapper assigns a unique random id for the output file such that no data is overwritten. BWA outputs sequence alignment data in Sequence Alignment/Map (SAM) format. The SAM/BAM Format Specification Working Group (2017) [8] specify that apart from header lines starting with `@`, 11 compulsory fields are found in each alignment line. These felds contain essential alignment information such as bitwise flag indicating whether the read mapped successfully or not and what type of mapping it is. This bitwise flag is specified in second column. Apart from the different values, there is one important value 4 which we need to discard since it means that a read is unmapped. A value of 0 means that a read maps to the positive DNA strand. Tang (2014) [76] gives a thorough explanation

[8] `https://samtools.github.io/hts-specs/SAMv1.pdf`

of bitwise flags also referring to an online tool[9] for converting bitwise flags to meaningful descriptions. If a read mapped successfully, mapping position, mapping quality and other optional fields are also included. Another crucial information in the SAM file is the *CIGAR* string which describes different types of alignments - match/mismatch/insertion/deletion/clipped/padded. Tang (2014) [76] explains how a *CIGAR* string is composed of a list of tuples containing alignment operation lengths and the actual operation. The basic *CIGAR* string caters for the three primitive operation types: M for match or mismatch, I for insertion and D for deletion.

- M - Sequence match or mismatch
- I - Insertions in aligned read
- D - Deletions in aligned read
- S - Soft clipping in aligned read
- H - Hard clipping in aligned read
- N - Skip in reference sequence
- P - Padded alignment

Example 3.5.1.

`30M6I51M13S`

means that for an input read of 100 base pairs we have, first 30 base pairs are matches or mismatches, then six insertions, 51 are matches or mismatches and final 13 base pairs are soft clipped

SAM tools [77] provides specialized utilities for efficiently parsing and manipulating SAM files. We managed to integrate SAM tools with Python using *pysam*[10],

[9] `http://broadinstitute.github.io/picard/explain-flags.html`
[10] `https://pysam.readthedocs.io/en/latest/api.html`

since SAM tools are originally written in C. *pysam* assumes that SAM tools are installed and pre-configured on each data node. An installation script is provided and explained in more detail in Appendix A. In the post-processing part, each mapper uses *pysam*, to parse BWA output data in SAM format in order to extract all the info explained in Section 3.3.1. First we check for any insertions by parsing the *CIGAR* string. If *I* is present, then all the *CIGAR* tuples in the current aligned segment are extracted. The insertions positions in the query sequence are extracted by going to the *CIGAR* tuple signalling an alignment match which comes exactly before the current *CIGAR* tuple signalling an insertion and adding the number of insertions to the last position match. The insertions read positions are stored in a temporary list for later usage.

Example 3.5.2. Using the *CIGAR* string

`30M6I51M13S`

then we can extract that the aligned read has insertions at positions 30, 31, 32, 33, 34, 35. Keeping in mind that positions are 0-based, hence the last match before the insertions is at position 29.

Each mapper has a key-value map containing all distinct matched reference genome positions together with the count of matched base pairs from the input reads. For each aligned base pair in the input read, the mapper extracts the reference genome position, the reference genome base pair at that position and the matching chromosome name. If the input read aligned position is not found, than it means we have a deletion in the input read, hence the count of *D* (deletions) at the current reference genome position in the key-value map is incremented. If the reference genome position is not specified and the current input read position is one of the insertions positions extracted previously, then it means we have an insertion at the current reference genome position. The insertions positions are added to the key-value map as decimal places (`.nn`) following the previous successful matched reference genome position, thus creating a pseudo-reference position. Hence we

know that at nth position after a particular reference genome position, there are insertions.

Example 3.5.3. Using the previously extracted insertion positions 30, 31, 32, 33, 34, 35, we extract base pairs from input reads at these positions and append them after the previous successful matched reference genome position, hence obtaining:

```
56776293        C       chr17   0,1,0,0,0,0
56776293.01     None    chr17   0,0,0,1,0,0
56776293.02     None    chr17   1,0,0,0,0,0
56776293.03     None    chr17   0,0,0,1,0,0
56776293.04     None    chr17   1,0,0,0,0,0
56776293.05     None    chr17   0,0,0,1,0,0
56776293.06     None    chr17   1,0,0,0,0,0
56776294        T       chr17   0,0,0,1,0,0
```

This means that at reference genome position 56776293, there is base pair `C` and input sequence read has `C` as well. Then there are six insertions (three times `T,A` respectively) in input sequence read, then at reference genome position 56776294, both reference genome and input sequence read have `T`. Using this approach the whole genome for the person in question can be created identifying also at which positions there are insertions

Each mapper outputs all elements in the key-value map where the key corresponds to the reference genome position whilst the value is a tuple containing three values: reference genome base pair at position specified in key, reference genome related chromosome name, combined counts of `ACGTDN` in input sequence read at reference genome position specified in key. These are streamed to the relevant combiner and eventually reducer. The combiner and reducer group data by reference genome position and aggregate counts for input sequence read base pairs. The final output is explained in Example 3.2. Figure 3.7 outlines the whole process flow.

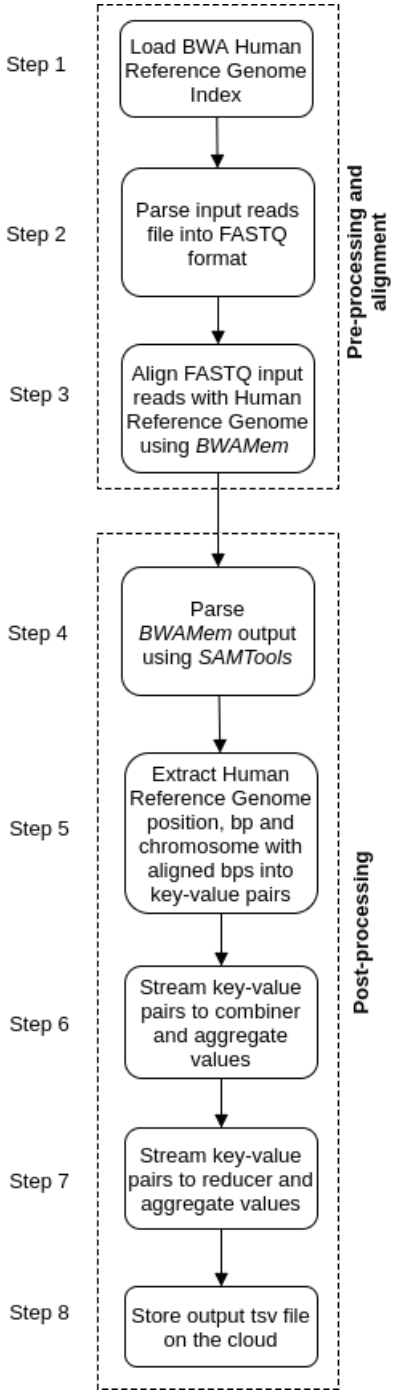


Figure 3.7: `MR-BWA` alignment process flow. The first three steps forming the pre-processing part are similar to `BigBWA`, whilst the post-processing part is a novelty in our approach

3.6 MR-BWT-FM

Our third approach uses heuristics to optimize suffix array creation and consequently FM-Index, based on the fact that genome has many repetitive sequences. The final aim is to reduce index size and lookup time. Similar to `MR-BWA`, these optimized data structures are then used for distributed genome sequence alignment based on MapReduce.

3.6.1 Suffix Array and BWT Optimization

The BWT algorithm (described in Section 2.3.5), which is the basis of the BWA software, is created via the suffix array (explained in Section 2.3.4). Creating suffix array for the whole reference genome consumes approximately 12GB [24]. BWA optimizes BWT by using a separate FM-Index as defined in Section 2.3.6. We opted to implement further optimizations as recommended in [78] based on the prefix doubling algorithm explained in Section 2.3.4 and Timsort algorithm explained in Section 2.3.7. Our function `text_to_int_keys(l)` is a crucial part of the optimized suffix array. This function extracts integer keys from a given iterable of keys belonging to a string suffix. After a few steps, this array is almost sorted since a further optimization is used through Timsort [38]. The Timsort works fine on repetitive data such as DNA and works best on data which is already partially sorted, like this case. This complements the principle of prefix doubling, which is precisely to use the partial order that is already computed. Louis Abraham[11] proposes a further optimisation in order to reduce memory consumption. This is based on the fact that in repetitive strings such as DNA, the last lines of the matrix are the same, hence these can be discarded. We implemented this heuristic in the main function responsible for the suffix array creation; `suffix_array(s)`. This function extracts all indices using the previous function `text_to_int_keys(l)` based on prefix doubling. Once all indices have been collected the suffix array is

[11] `https://louisabraham.github.io/notebooks/suffix_arrays.htm`

done. Eventually, the sorted indices of all suffixes are obtained by doing an inverse permutation of the suffix array. This leads to having $O(nlog^2n)$ time complexity. Although this function grows linearly using this optimisation, it still consumes a lot of memory because of the map and set data structures. The set contains each distinct character in input text whilst the map contains the character index for each distinct character in input text. Another optimization implemented is the downsampling of the suffix array as suggested by Langmead [79]. The idea presented is that of discarding most of the entries and re-creating them on the fly as needed whilst looking up offsets. Instead of storing every entry of the suffix array, every nth element is stored. Our function `downsample_suffix_array(sa, n=32)`, implements this idea by using a default of $n = 32$, that is, only the suffix array entries for every 32nd offset are considered. This saves a huge amount of disk space, which is explained in more detail in Section 4.5.

Once, the optimised suffix array has been created, BWT can be computed. This is explained in Section 2.3.5, however we implemented a further optimisation proposed by Langmead [79], based on the ranks for each character in LM Mapping. Pre-calculating the ranks arrays for the whole human genome is not practical, instead a similar approach to the suffix array downsampling is used. Instead of keeping all the ranking arrays from the computed BWT, the majority of these ranks are discarded, keeping only every nth rank. The retained rows are called rank checkpoints. This further reduces disk space consumption. Every time we need to look for row r, this mechanism works as follows:

- If row r was not discarded, usual lookup is performed
- If row r was discarded, characters starting at r in the last column are scanned forward/backward until the next checkpoint is reached. The resultant rank is equal to the rank of the reached checkpoint in addition to the number of steps involved to reach it from the discarded row.

Since checkpoints are spaced at a constant from each other, the time complexity

of this operation is $O(1)$. This is all implemented in `FmCheckpoints` class which manages rank checkpoints and handles rank queries. In this case the default chosen $n = 128$, that is, every 128th rank is stored.

3.6.2 Custom FM-Index Implementation

The complete FM-Index is made up of four components: the first column containing number of occurrences of each character, the BWT itself, the downsampled suffix array and the rank checkpoints. These are all computed in the `make_index(t)` function. This function is a wrapper function which calls all other previously mentioned functions to create suffix array, BWT and rank checkpoints. These four data structures are serialized and saved to disk using the Python `cpickle`[12] library. `cpickle` is fast because it is implemented in C. Moreover, data structures are saved in binary format to further improve on disk space utilisation. Searching for a given pattern is implemented in the `bwm_range(bwt_fmindex, query, mismatches)` function. This computes the ranks for the query using the `FmCheckpoints.rank()` function. This is based on the rank checkpoints procedure explained previously, whereby if a rank has been discarded, the count of steps to reach the next checkpoint is added. Once the BWM rows have been identified, the offsets for the query with respect to the reference genome are returned by the `resolve(bwt_fmindex, row)` function. For utility purpose, two different functions returning the occurrences of a pattern have been implemented:

- `all_occurrences(query, bwt_fmindex, mismatches=1)`: returns offsets for all occurrences of query based on the supplied BWT-FM. This caters for up to n mismatches, where by default $n = 1$
- `first_occurrence(query, bwt_fmindex, mismatches=1)`: returns the offset for first occurrence of query based on the supplied BWT-FM. This caters for up to n mismatches, where by default $n = 1$

[12] `https://docs.python.org/2/library/pickle.html#module-cPickle`

Figure 3.8 outlines all the steps involved in creating the custom FM-Index.

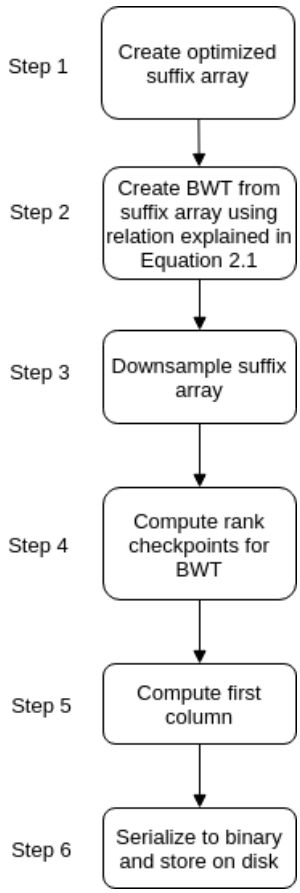


Figure 3.8: Custom FM-Index process flow. The creation of BWT from suffix array in Step 2 is based on Equation 2.1

3.6.3 BWT-FM Distributed Alignment

Similar to what has been explained in Section 3.5.2, input `FASTQ` files are again transformed to Hadoop format whereby every four lines in `FASTQ` files representing a read are converted to one line. The preprocessed input files are uploaded to HDFS to act as input for mappers.

Each mapper is divided into pre-processing and post-processing as shown in Figure 3.9. First, it checks that a valid FM-Index associated file exists on the associated data node and loads it using `cpickle` library. If file is not found, the mapper halts immediately. Eventually, the mapper parses the preprocessed input

streamed from HDFS and aligns each read with the loaded FM-Index using the `first_occurrence(query, bwt_fmindex, mismatches=1)` function. In this case, the first occurrence is enough in order to get the initial reference genome position. Similar to Section 3.5.2, each mapper has a key-value map containing all distinct matched reference genome positions together with the count of matched base pairs from the input reads. The reference genome base pair, together with the query base pair are extracted. Mismatches are computed based on the Smith-Waterman algorithm, which also handles insertions and deletions. The mapper output is identical to the one in Section 3.5.2. This was crucial, since we want to compare both approaches and have a final output file which does not vary in format.

Each mapper outputs all elements in the key-value map where the key corresponds to the reference genome position whilst the value is a tuple containing two values: reference genome base pair at position specified in key, combined counts of `ACGTDN` in input sequence read at reference genome position specified in key. These are streamed to the relevant combiner and eventually reducer. The combiner and reducer group data by reference genome position and aggregate counts for input sequence read base pairs. The final output is explained in Example 3.2. Figure 3.9 outlines the whole process flow.

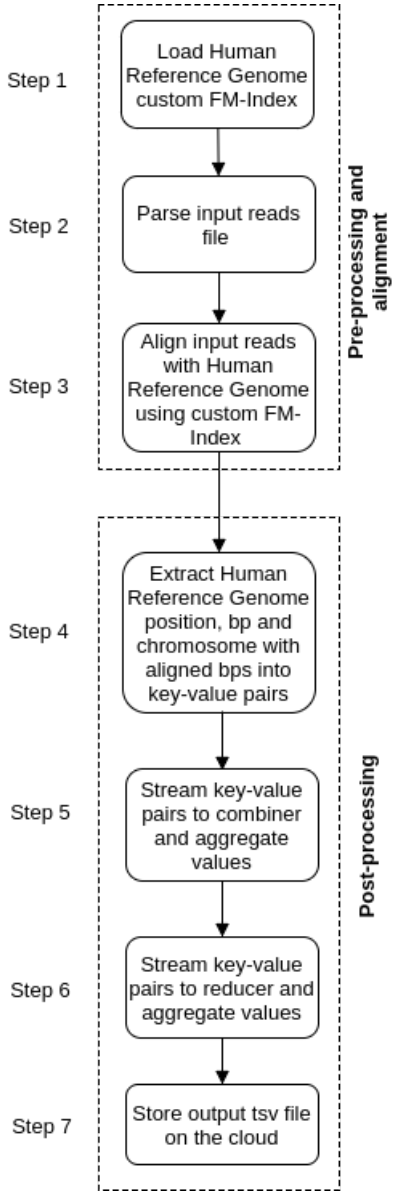


Figure 3.9: `MR-BWT-FM` alignment process flow. This is similar to `MR-BWA` as explained in Figure 3.7.

4. Results and Evaluation

We evaluate our approach with an industry standard alignment tool, BWA, for both speed and correctness. We also compare from literature our approach to a similar approach taken by Abuin et al. [4]. An Apache Hadoop Cluster is set up on Amazon EC2, identical to the one used in `BigBWA` by Abuin et al. [4]. Moreover, the same datasets obtained from the 1000 Genomes Projects [44] have been used for speed benchmarking. Alignment correctness for the second and third approach, `MR-BWA` and `MR-BWT-FM`, is only compared to the standard BWA since Abuin et al. [4] do not report results in relation to alignment correctness. For the alignment correctness part, we implemented an `Rscript` as explained in Section 3.2.1. The software used for evaluation is outlined in Table 4.1

4.1 Cloud Infrastructure

An Apache Hadoop cluster on Amazon EC2 cloud[1] has been set up to evaluate our system. The system has been successfully tested also on Microsoft Azure cloud[2], however Amazon has been finally chosen in order to be comparable, in terms of hardware, with evaluations done by Abuin et al. [4]. In our case, the Hadoop virtual cluster consists of five nodes of the `r3.4xlarge` instance type, identical

[1] `http://aws.amazon.com/ec2/`
[2] `https://azure.microsoft.com/`

Name	Author(s)	Description
`BWA`	Li and Durbin[9]	Industry standard alignment tool BWA as explained in Section 2.4. Whenever there is a mention in format `BWA-n Threads`, this means that a multi-threaded version of `BWA` using n threads has been executed. In all cases, the `BWA-MEM` has been used as it is faster and more accurate. Moreover our implemented approaches explained in Chapter 3, both use the `BWA-MEM` algorithm.
`MR-BWA`	Karl Pullicino (extending upon Li and Durbin[9])	Distributed BWA version as explained in Section 3.5.2
`MR-BWT-FM`	Karl Pullicino	Distributed custom version of BWT and FM-Index as explained in Section 3.6.3. The `BWT-FM` is the same version however using only one core
`BigBWA`	Abuin et al. [4]	BigBWA application explained in Section 2.4.1

Table 4.1: Tools used for evaluation

to the ones used in [4]. Amazon specify that such instances are optimized for memory-intensive applications. Moreover, these are built specifically for *"genome assembly and analysis"*[3]. The specifications for each node are as follows:

- Operating System: Ubuntu, 16.04 LTS, amd64
- CPU: Intel Xeon E5-2670v2 2.5 GHz (Ivy Bridge)
- Cores: 16

[3] `https://aws.amazon.com/ec2/instance-types/`

- RAM: 122GB
- Disk: 320GB SSD
- Network: High throughput between nodes

An overview of the whole setup is shown in Figure 4.1

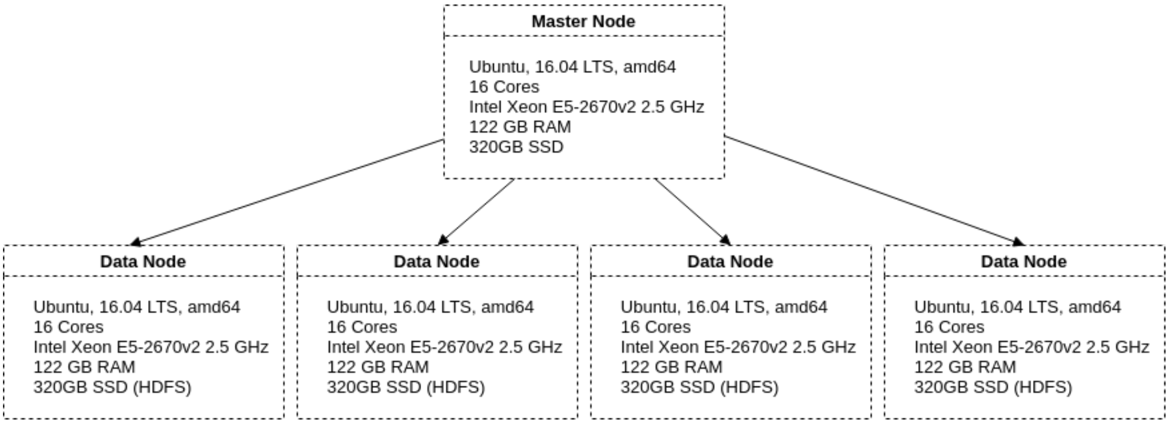


Figure 4.1: AWS Apache Hadoop cluster architecture

Apache Hadoop v2.7.1[4], Python v2.7.6[5], BWA v0.7.15[6] and SAM tools v1.4.1[7] have been installed on each AWS node. MapReduce jobs were executed using the Hadoop streaming API v2.6.5[8]. Installation and setup procedure is explained in detail in Appendix A.

Both MapReduce and HDFS are based on a master/slave architecture as explained in Section 2.2. The master node is used to launch the Hadoop jobs and manage HDFS. The data nodes perform the map and reduce operations. In total, we had 64 computing cores and 488GB dedicated to map and reduce processes. The HDFS block size is 128MB, meaning that a map operation is spawned for every 128MB of data that is being processed. The memory size assigned to map and reduce tasks is crucial in obtaining the best results. Since each data node

[4]https://hadoop.apache.org/docs/r2.7.1/
[5]https://www.python.org/download/releases/2.7.6/
[6]https://sourceforge.net/projects/bio-bwa/files/
[7]https://sourceforge.net/projects/samtools/files/samtools/1.4.1/
[8]https://hadoop.apache.org/docs/r1.2.1/streaming.html

loads the reference genome index in memory, which is approximately 6GB, the amount of memory for each map/reduce task has been set to 12GB. The extra 6GB were assigned to cater for loading the input reads in memory and processing the alignment output in order to extract the data needed as explained in Chapter 3.

Assigning memory per task beforehand limits the scalability of the Hadoop application. In fact, since in this case 12GB per map/reduce task were assigned, a maximum of 10 concurrent tasks can be executed concurrently on each node, since these will consume the whole 122GB RAM per node. This means that the 16 cores per node were not always being used concurrently.

4.2 Evaluation Datasets

Two datasets from the 1000 Genomes Projects [44] have been used to evaluate the two approaches mentioned in Chapter 3. These two datasets have also been used by [4] which helps us make a literature comparison with their approach. The input reads used are `HG00096/SRR062634`[9] and `NA12750/ERR000589`[10]. Details about these datasets are found in Table 4.2

Name	**Number of reads**	**Read length**	**Size (GB)**
`HG00096/SRR062634`	24,148,993	100	13.4
`NA12750/ERR000589`	11,964,008	51	3.9

Table 4.2: Evaluation datasets characteristics

Human reference genome hg38 (GRCh38 Genome Reference Consortium Human Reference 38)[11] has been used for all alignments.

[9] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00096/sequence_read/`
[10] `ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12750/sequence_read/`
[11] `http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/`

4.3 Alignment Performance Results

We compare the speed of our two approaches, `MR-BWA` and `MR-BWT-FM`, with `BigBWA` [4]. These performance results and comparisons take only into consideration the pre-processing and alignment part of each `mapper` function, that is, until the third step, as explained in Figures 3.7 and 3.9. Other tasks executed in post-processing stage of the `mapper` function such as parsing alignment output, extracting the reference position and aligned base pairs positions, are not considered in this case since these were not part of `BigBWA`. The post-processing part is discussed later. Data values were calculated as the mean time taken of three separate runs. For each run the tools described in Table 4.1, the cluster described in Section 4.1 and datasets mentioned in Section 4.2, have been used.

`MR-BWA` is more complete than the approaches explained by Al-Absi et al. [8] and Algur & Sakri [70, 71], since in our case the whole human genome having three billion base pairs is used as reference, whilst in their case only a subsection of the reference genome consisting of one million base pairs was used. In fact the Hadoop cluster used by [8, 70, 71], just has 7GB of RAM and four cores, which is not capable of handling the whole reference genome and the datasets mentioned in Section 4.2.

When comparing with `BigBWA` [4], both our approaches, `MR-BWA` and `MR-BWT-FM` have the advantage of pre-processing the reference genome index offline and store it on disk on each node, prior to starting any alignment. In our case, the reference genome index is created on one node and then transferred to all other nodes using the high bandwidth network available between nodes. This makes our approaches more cost effective, since we are saving on computation time from three data nodes. These costs could not be compared with `BigBWA` since Abuin at al. [4], do not make any reference to the bootstrap costs, e.g. creation of indices, incurred in obtaining their results. Moreover, any comparison done with `BigBWA` is solely a literature comparison with the results presented in [4], since this software is not regularly maintained.

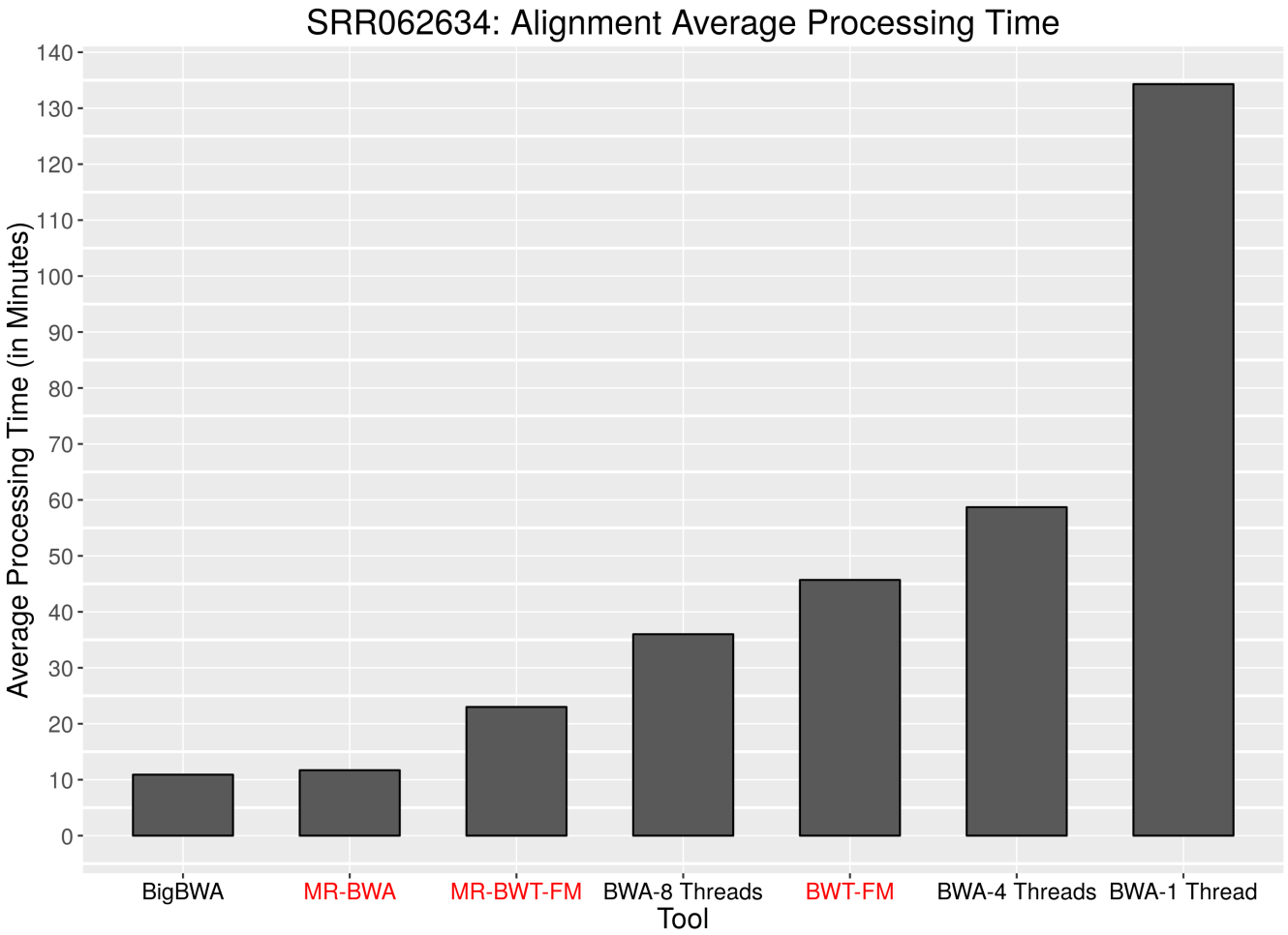


Figure 4.2: `SRR062634` dataset processing time. This shows the mean time taken of three separate runs for each tool using the `SRR062634` dataset. The tools marked in red are the ones which we implemented as described in Chapter 3. The top three ranked tools are all based on a Hadoop cluster, whilst the remaining four tools were executed on a single machine.

Figures 4.2 and 4.3 show the processing time in seconds taken by different tools when aligning reads to a reference genome. We note, that the tools ranked in the same order for both datasets, although there is a difference in dataset size and number of reads. Tools marked in red are the ones which we implemented as described in Chapter 3. `BigBWA` is the best performing tool for both datasets with `MR-BWA` slightly slower in both cases. This could be related to some optimizations used by `BigBWA`, since as explained in [4], the authors take advantage of the Java

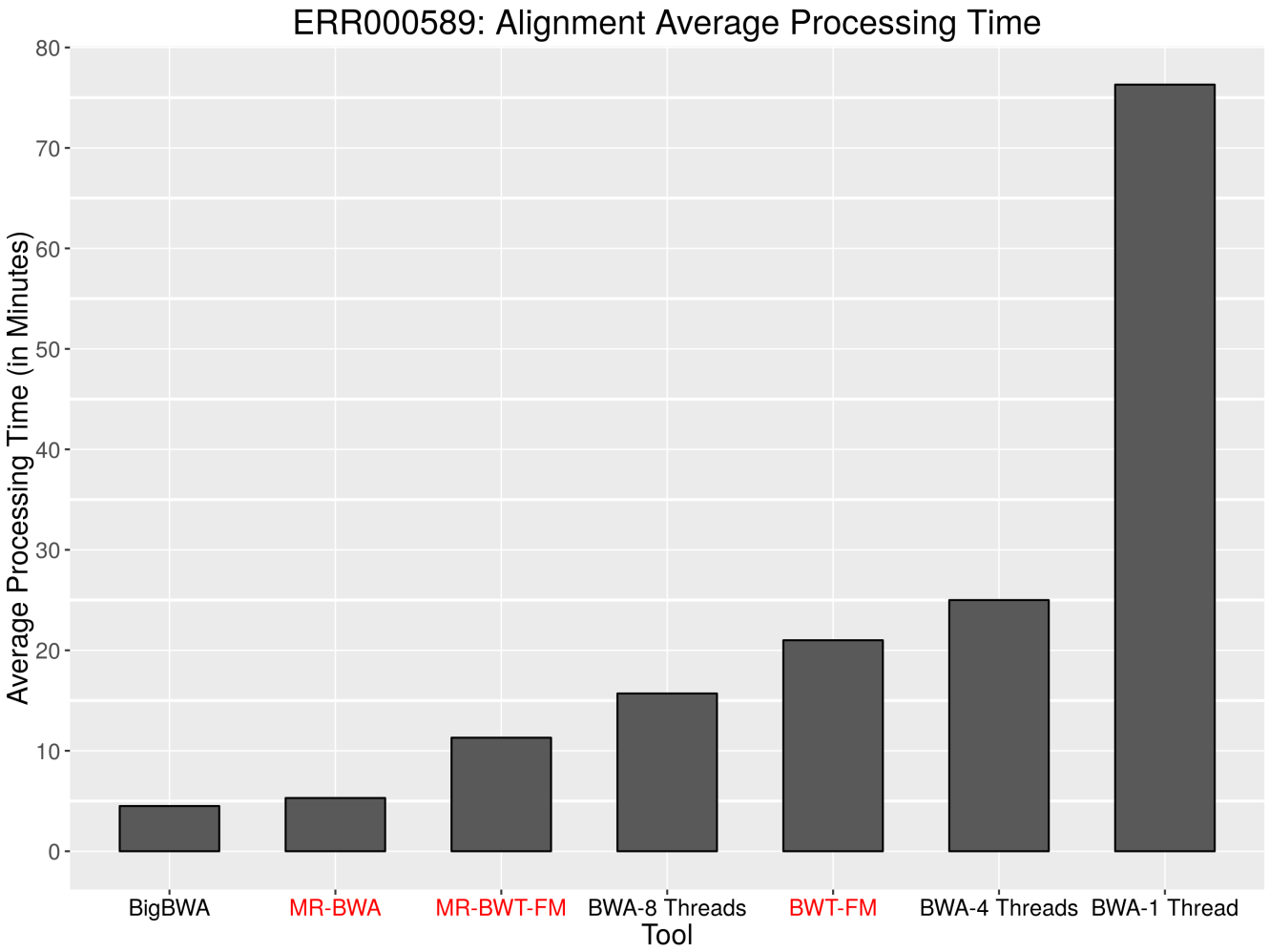


Figure 4.3: `ERR000589` dataset processing time. This shows the mean time taken of three separate runs for each tool using the `ERR000589` dataset. The tools marked in red are the ones which we implemented as described in Chapter 3. Results are similar to Figure 4.2. The top three ranked tools are all based on a Hadoop cluster, whilst the remaining four tools were executed on a single machine.

Native Interface (JNI)[12], which allows to include native code written in low-level programming languages such as C and C++, as well as code written in Java. When using dataset `SRR062634`, which has a size of 13.4GB and is composed of approximately 24 million reads, `BigBWA` is approximately 7% faster than `MR-BWA`. When using dataset `ERR000589`, which has a size of 3.9GB and is composed of approximately 12 million reads, `BigBWA` is approximately 18% faster than `MR-BWA`.

[12] `https://www.fer.unizg.hr/_download/repository/jni.pdf`

This means that MR-BWA performs better on larger datasets. This is probably due to the fact that MR-BWA takes some time to setup internal data structures related to the post-processing phase, during ramp up. In the long run, this helps achieve better results on larger datasets.

Both MR-BWA and MR-BWT-FM perform better than the BWA threaded version, even when using BWA with eight threads on multiple cores. MR-BWA is approximately twice as fast as MR-BWT-FM. The main reason for this, is the fact that the underlying optimizations used in the industry standard BWA software, are part of MR-BWA but are missing in MR-BWT-FM, since in the latter we built everything from scratch. The BWT-FM using one thread performed significantly better than the BWA counterpart with one thread.

The alignment processing time for each map task has been evaluated in comparison with the total map task processing time. As explained in Sections 3.5.2 and 3.6.3, the mapper is made up of a pre-processing phase and a post-processing phase. When the alignment has finished, the mapper performs the post-processing stage by parsing the alignment output in order to extract reference genome and aligned base pairs positions. This is something novel in our approach, which is not performed in other applications. From Figure 4.4 we note that approximately 94% of the total mapper task is spent parsing the alignment output in order to extract the reference genome position and the aligned base pairs positions. This is a bottleneck in our system which needs further improvement as explained in Section 5.2.

Data for Figure 4.4 has been extracted using MR-BWA tool to align the SRR062634 dataset. We note that in this case, the MapReduce job performed 50 map tasks. Both total mapper processing time and alignment processing time were uniformly distributed across all map tasks. This means that our MapReduce job did not suffer from straggler tasks. This is clearly important in a pay-by-the-hour environment like AWS since we are limiting costs. Straggler tasks are long duration tasks which effect the performance of the whole MapReduce job since the master node has to wait for such tasks to finish before assigning other tasks using the scheduler.

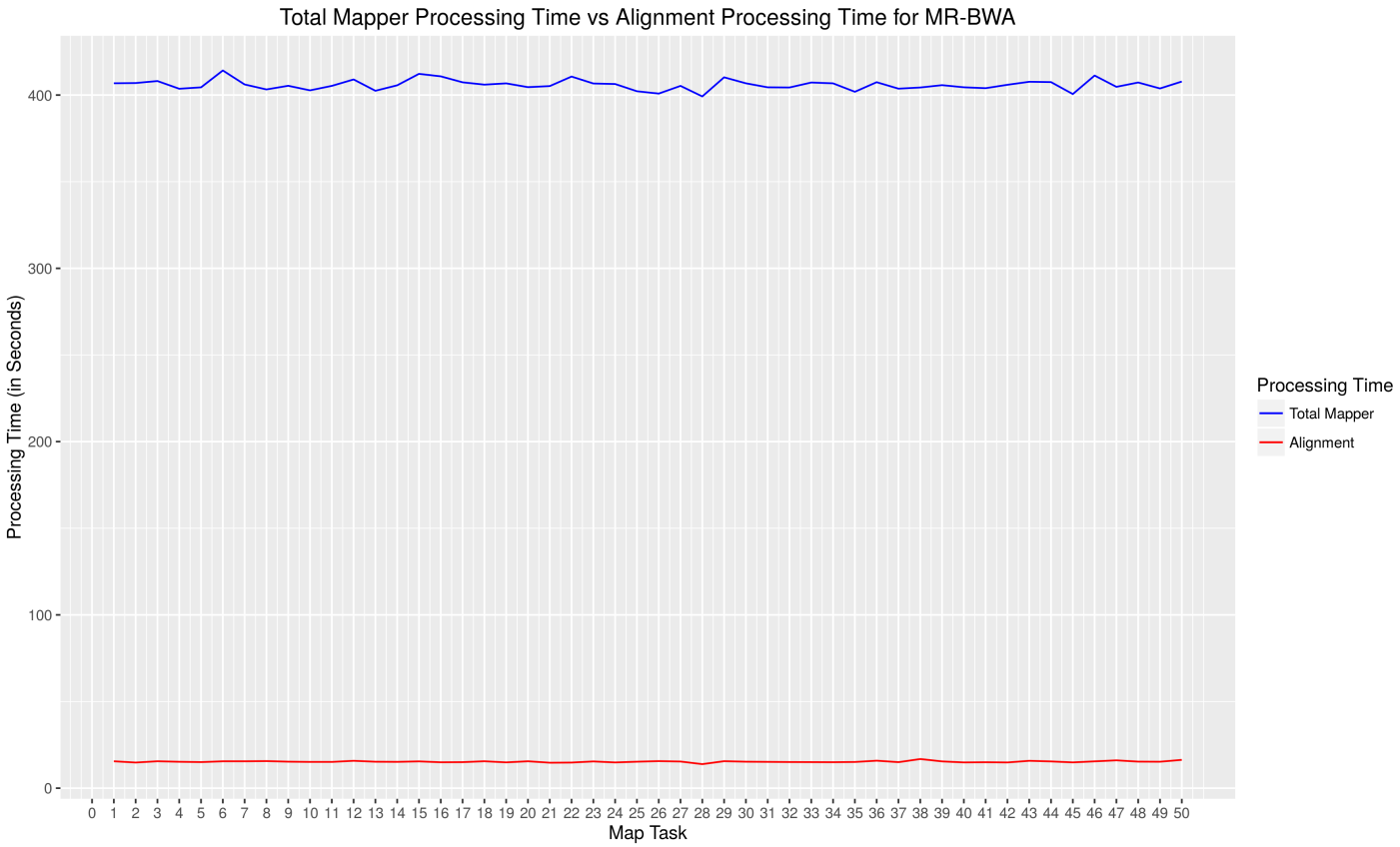


Figure 4.4: Total mapper processing time vs alignment processing time for `MR-BWA` tool to align the `SRR062634` dataset. In total, 50 map tasks were performed. The red line shows the pre-processing and alignment time for each map task whilst the blue line shows the whole mapper task processing time including pre-processing, alignment and post-processing stages.

4.4 Alignment Correctness

Alignment correctness was verified by comparing the output of dataset `HG00096/SRR062634` for three different tools: `BWA`, `MR-BWA` and `MR-BWT-FM`. `BigBWA` was not considered since the authors do not report results with regards to alignment correctness. They just mention that the differences between BWA and `BigBWA` range from 0.06% to 1% on uniquely mapped reads. In our case, the `RScript` output analysis explained in Section 3.2.1 has been used.

The first correctness metric is based on the alignment operation distribution per tool. The idea is to compare the count of base pairs for each tool into different

categories:

1. Match: when the reference genome base pair and input read base pair match at a specific reference genome position

2. Variation: when the reference genome base pair and input read base pair do not match at a specific reference genome position

3. Insertion: refers to an insertion in the provided sample at a specific position

4. Deletion: refers to a deletion from the provided sample at a specific position

Alignment Operation	`BWA`	`MR-BWA`	`MR-BWT-FM`
Match	57.91%	58.11%	55.45%
Variation	42.05%	41.85%	44.5%
Insertion	0.02%	0.02%	0.02%
Deletion	0.02%	0.02%	0.02%

Table 4.3: Alignment operation distribution for different tools using `SRR062634` dataset

Table 4.3 shows that the differences between `BWA` and `MR-BWA` are in the range of 0.2% for both matches and variations. However, there is a difference of approximately 2.5% between `MR-BWT-FM` and both the other tools. `MR-BWT-FM` shows an increase in variations which directly implies a decrease in the number of matched base pairs. The higher number of variations in `MR-BWT-FM` comes from the fact that the custom built algorithm catering for mismatches needs to be tweaked. The percentage of insertions and deletions for all three approaches is practically the same. The alignment operation distribution for the three tools is shown graphically in Figure 4.5

The second correctness metric used was to analyze the distribution of variations per chromosome per tool. The idea is to infer that all tools are consistent in which chromosomes, the highest number of variations exist. Again, dataset

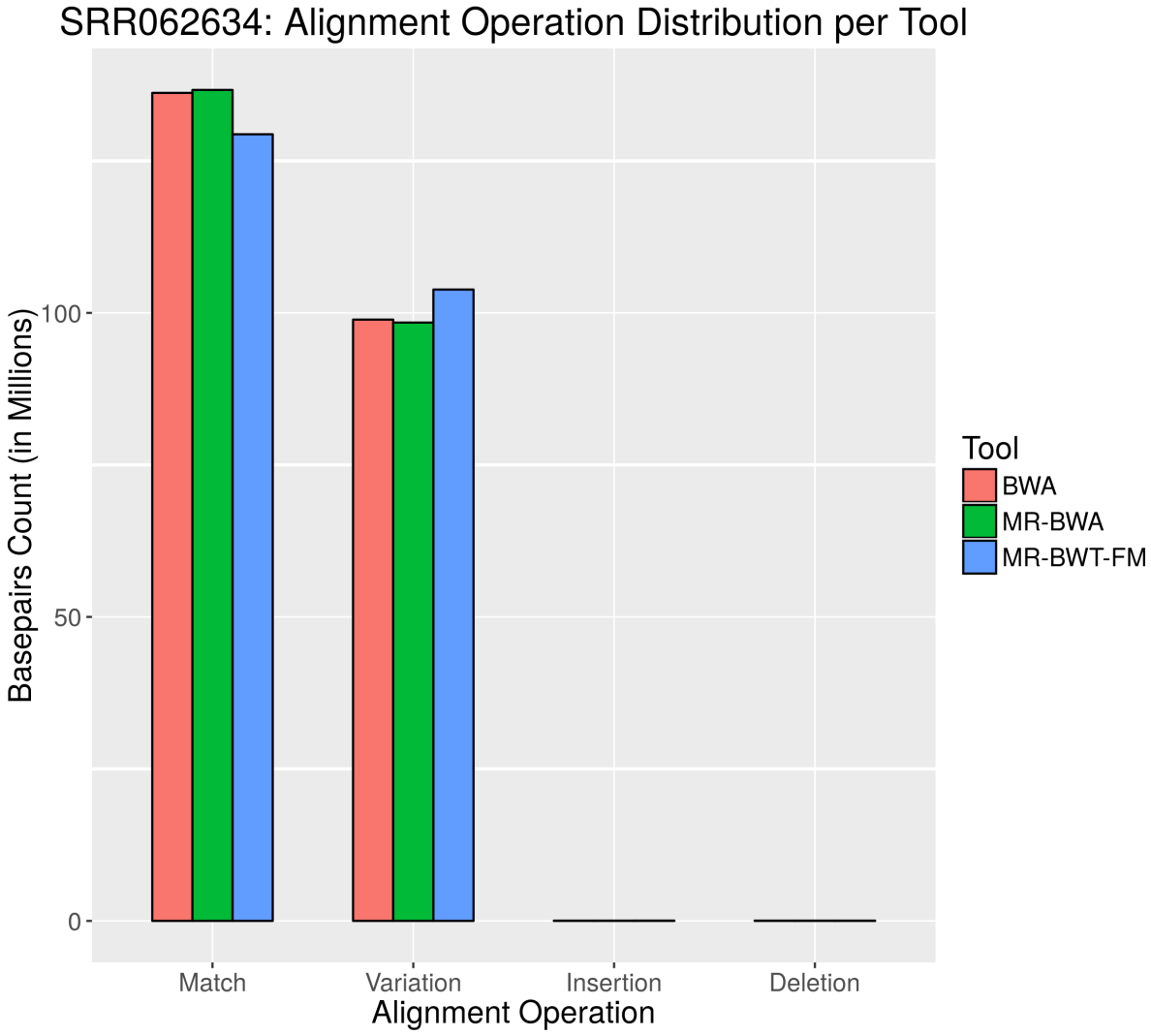


Figure 4.5: Alignment operation distribution per tool using `SRR062634` dataset. The output for `BWA`, `MR-BWA`, `MR-BWT-FM` to align the `SRR062634` dataset has been parsed and the count of base pairs has been grouped by alignment operation type. This shows that `BWA` and `MR-BWA` are similar having the same distribution. `MR-BWT-FM` shows a higher number of variations.

HG00096/SRR062634 has been used to analyze the number of variations for the three different tools: BWA, MR-BWA and MR-BWT-FM. In this case, only the chromosomes having total variations greater than the third quartile have been considered. We deemed that it is not worth pointing out any chromosomes having total variations less than third quartile since these have a minor effect on the whole genome. As we can observe in Figure 4.6, the distribution is the same for all tools, however the chromosomes vary from tool to another. Case in point is that whilst chr8 tops the number of variations for BWA and MR-BWT-FM, it is not present in MR-BWA. chr7 is present in all three tools. Although the two longest chromosomes, chr1, chr2, are not present in any distribution shown in Figure 4.6, the chromosomes shown refer to next longest chromosomes such as chr3, chr4, chr5, chr6, chr7. A full list of human chromosomes is available in Ensembl genome browser[13].

The analysis tools presented here should help researchers find interesting mutations for a specific genome.

[13] http://www.ensembl.org/Homo_sapiens/Location/Genome?r=Y:1-1000

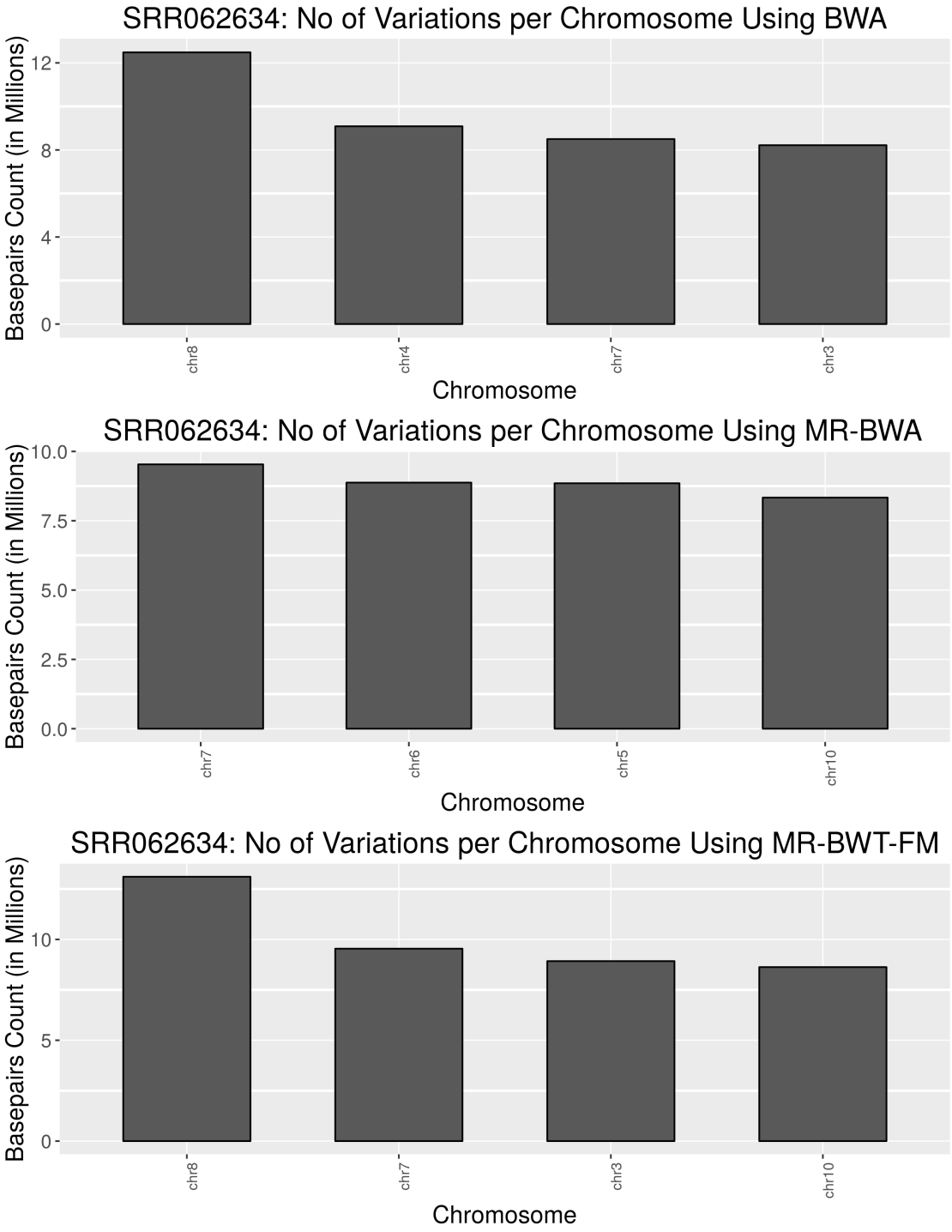


Figure 4.6: Variations distribution per chromosome per tool using `SRR062634` dataset. The output for `BWA`, `MR-BWA`, `MR-BWT-FM` to align the `SRR062634` dataset has been parsed and only the variations have been considered. The count of base pairs have been grouped by chromosome in order to verify that all tools report the same chromosomes having most variations.

4.5 Custom FM-Index Optimizations

The final section that was evaluated is in relation to the FM-Index optimizations explained in Section 3.6. Using the suffix array down-sampling and FM checkpoints, we saved on disk size. In fact as shown in Figure 4.7, the `Custom FM-Index` is 15% smaller in disk size. Reference genome index size using `Custom FM-Index` is 4.65GB, whilst reference genome index size using `BWA Index` is 5.36GB. However, the `Custom FM-Index` is approximately 18 times slower than the `BWA Index`. Indexing the reference genome using `BWA Index` took approximately 90 minutes, whilst indexing the reference genome using `Custom FM-Index` took 27 hours. Moreover, `Custom FM-Index` was executed using a memory-optimized VM having 20 Cores and 140 GB memory, whilst `BWA Index` used an 8-core and 16GB laptop. The main drawback of `Custom FM-Index`, is the huge amount of memory needed to create the suffix array. This needs to be further improved as mentioned in Section 5.2. Although, the `Custom FM-Index` is slow, we have to keep in mind that the reference genome index is created only once offline. So, given the right optimizations in suffix array, it may be worth taking more time once as long as we are saving on disk size.

Human reference genome hg38 (GRCh38 Genome Reference Consortium Human Reference 38)[14] has been used to evaluate the indexing scenarios mentioned above.

[14] `http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/`

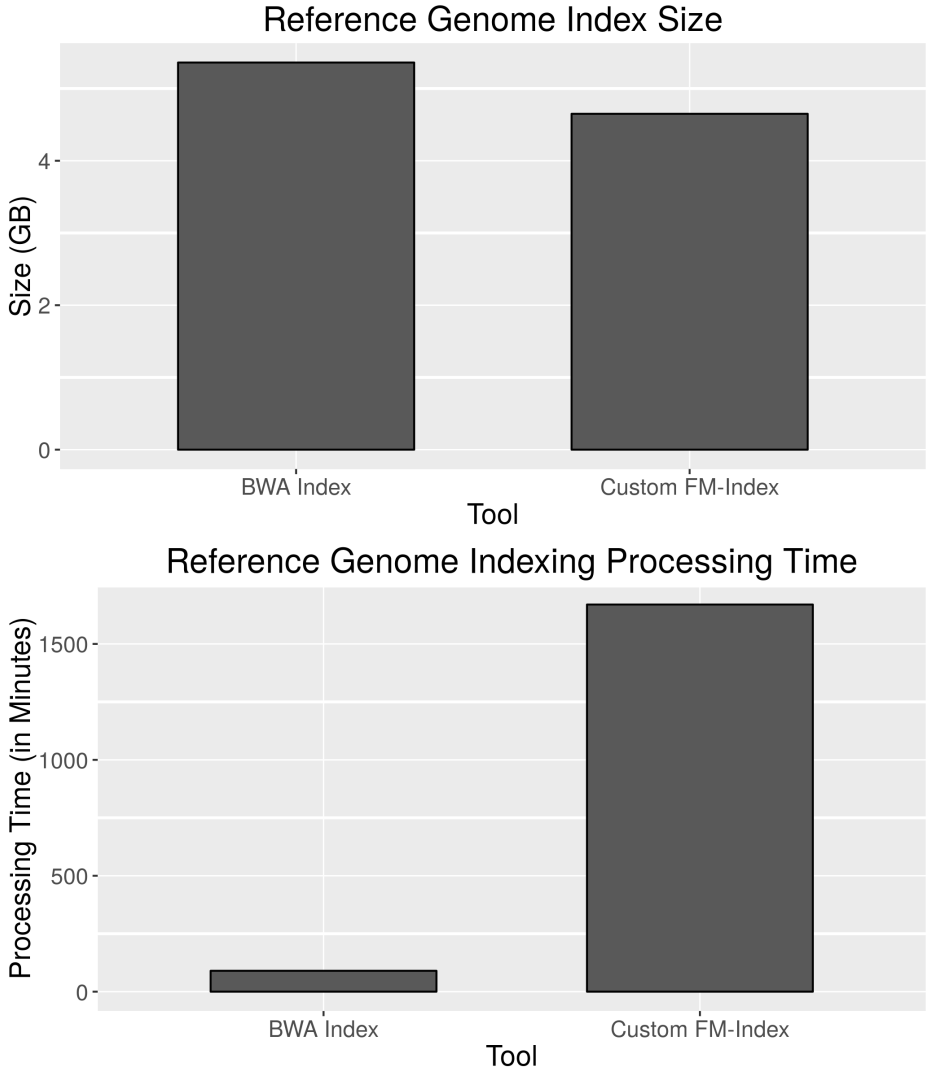


Figure 4.7: Reference genome indexing metrics comparing reference genome index size and processing time using the industry standard `BWA` and `Custom FM-Index`

5. Conclusions

This chapter presents a summary of our dissertation. Furthermore, we discuss the extent to which the objectives of the dissertation have been met. A section dedicated for any possible future work is also presented.

Chapter one, presents the motivation behind this dissertation, that is, finding the optimal way of aligning many short DNA reads from a biological sample to a reference human genome so that the DNA of the sample can be assembled. The DNA alignment problem and the huge amount of data generated in DNA sequencing are also presented. We present our aims and explain how we can use distribution to tackle the problem.

Chapter two, describes the background material required to understand this dissertation and discusses related work in the field. Given the huge data size involved in this area, big data paradigms and cloud computing are used. We present, MapReduce, a commonly used cloud computing framework for massive computing in the cloud and the main technology behind this dissertation. String related algorithms and indices construction techniques are also presented. The chapter ends with a discussion about related work, with the main focus being distributed alignment techniques.

Chapter three, outlines the design and implementation of our dissertation. Our three original approaches are presented. The first one, based on an optimization of the Smith-Waterman algorithm, did not obtain desired results due to memory

constraints. For this reason, the main focus was given to the other two approaches; `MR-BWA` and `MR-BWT-FM`. These are based on MapReduce programming paradigm in order to perform distributed sequence alignment of genomic data. The first one is based on a distributed version of the BWA which has been integrated into our Python workflow, whilst the second, presents an optimized version of the FM-Index in our distributed alignment implementation.

Chapter four, presents the results obtained by our approaches are evaluated and compared with `BigBWA` using datasets from the 1000 Genomes Projects [44]. Our work is evaluated for both alignment performance and correctness. The custom FM-Index built in `MR-BWT-FM` is compared with the BWA index in terms of processing time and disk size.

5.1 Objectives Achieved

The main aim of this dissertation was to provide a cloud-based implementation performing distributed sequence alignment of genomic data. The performance of such system should be comparable with similar systems. Also, the result of this cloud-based system should be easily used to reassemble the human genome of the sample and compare it with the reference human genome. This output, can be used to analyze variations with the reference genome which may cause genetic diseases. In general, this goal has been achieved as shown in Chapter 4.

The first objective set, was to parallelize the Smith-Waterman algorithm. Such objective was unsuccessful. We managed to implement an optimization by choosing the best alignment based on the best base quality and surrounding context, rather than taking the first maximum score. However, the parallelization part was not implemented since when trying to align the input reads with the whole reference genome, we ended with out of memory errors as explained in Section 3.4. Possible improvements for this approach are proposed in Section 5.2.

Given that the previous objective related to SW algorithm parallelization was

not fully successful, we focused on the next aim, to distribute sequence alignment using MapReduce programming paradigm. In this case, the objective has been met by integrating BWA with Python and implementing an artifact performing distributed sequence alignment using MapReduce. Alignment performance results are comparable with similar systems. As shown in Section 4.3, our approach `MR-BWA` is approximately 7% slower than `BigBWA` using a 13.4GB input reads file. However it was noted that as the input file gets larger, `MR-BWA` reduces the performance gap with `BigBWA`. Moreover, `MR-BWA` is based on a novel approach of not splitting the reference genome, instead only the input read files are distributed. Based on the results obtained, it is clear that this approach has been successful. `MR-BWA` exploits the advantages of any MapReduce based system, that is, distributed computation, leading to reducing execution times and also the fault tolerance capabilities of the underlying Hadoop technology. Moreover, no modifications to the original BWA have been made, making `MR-BWA` working with any version of BWA. `MR-BWA` presents also an innovative output format which could help researchers to analyze genomic variants as explained in Section 3.2.1.

The final aim was to optimize suffix array creation and consequently FM-Index, based on the fact that the genome has many repetitive sequences. This has been achieved since reference genome index created using the `Custom FM-Index` techniques is 15% smaller in disk size than the reference genome index size created using the standard `BWA Index`. The main drawback in this case is related to the fact that the created `Custom FM-Index` is 18 times slower than the `BWA Index`. Improvements for this are proposed in Section 5.2, however this is not considered an issue since index for the reference genome does not change and is created once offline. The `MR-BWT-FM` approach using the custom FM-Index built from scratch, performed well, ranking third in the overall performance analysis as shown in Section 4.3. However, `MR-BWA` is approximately twice as fast as `MR-BWT-FM`, hence the best idea would be to incorporate our custom FM-Index with `MR-BWA`, possibly reducing `MR-BWA` ramp-up time since less memory is needed to load reference genome index.

5.2 Future Work

Although, the main objectives have been achieved successfully, all three approaches presented in Chapter 3, have room for improvement.

The custom SW algorithm should be parallelized in a similar way proposed by Venkatachalam [66]. The idea is to compute the matrix entries in parallel by exploiting the fact that although all the elements of the anti-diagonal depend on the previous anti-diagonal, they are independent of each other. Trelles [80] presents an idea to have one of the processors acting as a "master", dispatching blocks of sequences to the "slaves" which, in turn, perform the calculations. When a slave has finished computation for one block, the master sends data to a new block. This is possible since results from the comparison between query and reference sequences are independent of the previous results coming from the comparison of the query with other reference sequences. Computation time depends solely on the sequence, but not all sequences have the same composition hence a load balancer would be needed whereby the master distributes the load evenly amongst slaves. Instead of distributing sequence-by-sequence, better results are achieved by using blocks of sequences.

The main area requiring improvement in `MR-BWA` is related to the fact that approximately 94% of the total mapper task is spent parsing the alignment output in order to extract the reference genome position and the aligned base pairs as explained in Section 4.3. Currently, the computations to extract the reference genome position and the aligned base pairs are performed sequentially after the alignment for the whole input file has finished. These computations could be done in parallel with the alignment, using a concurrent hash table per map task that is continuously updated. Moreover, rather than processing output alignment for each read in a sequential manner, the extraction of reference genome position and the aligned base pairs should be multi-threaded per map task. This is possible via a specific multi-threaded mapper construct. Using such approach, the whole map task processing time is reduced, hence gaining in the global processing time.

`MR-BWA` should be adapted to be compatible with the `Custom FM-Index` explained in Section 3.6.2. Since the custom FM-Index has a smaller disk size, this means that `MR-BWA` would take less time to ramp-up since less memory is required to load the reference genome index. This will help achieve better performance results even on smaller files to those explained in Chapter 4. Further research should be done in order to optimize MapReduce configuration in order to have the minimum idle time as possible in the map/reduce tasks. From an end user's perspective, it is better to create a Python module to be used by general public by deploying it to Hadoop, instead of going through all the steps of setting up and running the application as explained in Appendix A.

Although the `MR-BWT-FM` approach achieved good results, there are various areas that may be improved. The core of the index optimizations is based on the suffix-array creation. We managed to reduce index size at the expense of increasing processing time to create the index. The majority of the time taken is to create the suffix array for the whole reference genome. A possible improvement is to parallelize the suffix array creation. Nahar and Tseng [81] propose two different algorithms for suffix array construction using distributed cluster computing. The first one called *pDC3* is a classic suffix array construction algorithm adapted for cluster computing, whilst the second algorithm *pCSS*, is a novel algorithm they developed having low inter-node communication complexity and scaling well across cores.

As shown in Section 4.4, `MR-BWT-FM` has a difference of approximately 2.5% in alignments distribution when compared with both `BWA` and `MR-BWA`. `MR-BWT-FM` shows an increase in variations which is mainly the side-effect of a non-optimal approximate matching algorithm used when aligning input reads with the custom FM-Index. The approximate matching algorithm should be improved and based on the pigeonhole principle as explained in Section 2.3.8. This should be optimized and work in a similar fashion to edit distance approach described in Section 2.3.9. A further improvement for the `MR-BWT-FM` approach is to refine the function used to get best aligned read. This function should return the maximum score based on the

best base quality and surrounding context, rather than taking the first maximum score. The idea is that neighbouring reads to the chosen best read, should also have scores as close as possible to the highest score.

From an evaluation dataset perspective, the original idea of this dissertation was to use Maltese genome data for evaluation purposes, in order to extract any interesting mutations related to the Maltese genome. In our case, the initial Maltese samples were sequenced using a proprietary technology which is incompatible with BWA and hence we had to resort in using 1000 Genomes Project data for evaluation purposes. Further work in this area would include, adapting `MR-BWT-FM` to load these proprietary reads.

5.3 Final Words

This dissertation presents a cloud-based implementation performing distributed sequence alignment of genomic data. Two approaches based on industry standard alignment tools such as BWA and other custom index optimizations are implemented. Both approaches achieve good results and offer contributions towards the scientific community. Further improving the implementation with the suggestions explained in Section 5.2, will improve the final contributions of this dissertation.

A. Setting up MR-BWA on Hadoop

This appendix provides a guide to setup and run `MR-BWA`. It is assumed that an Apache Hadoop Cluster has already been setup as described in Section 4.1

1. Apache Hadoop Cluster configuration is updated to have specific memory assigned:
   - `yarn-site.xml`
     - `yarn.log-aggregation-enable=true`
     - `yarn.nodemanager.pmem-check-enabled=false`
     - `yarn.nodemanager.vmem-check-enabled=false`
     - `yarn.nodemanager.resource.memory-mb=12288`
     - `yarn.scheduler.maximum-allocation-mb=12288`
   - `mapred-site.xml`
     - `mapreduce.map.memory.mb=12288`
     - `mapreduce.reduce.memory.mb=12288`
     - `mapreduce.map.java.opts=-Xmx12288M -Djava.net.preferIPv4Stack=true -XX:NewRatio=8 -XX:+UseNUMA -XX:+UseParallelGC`

- `mapreduce.reduce.java.opts=-Xmx12288M -Djava.net.preferIPv4Stack=true -XX:NewRatio=8 -XX:+UseNUMA -XX:+UseParallelGC`

2. Upload and run the provided `setup_biotools.sh, hd_biotools.sh` on each node. These scripts will install and setup Python, BWA and SAM tools. Any extra Python dependencies are also installed.

3. Upload the provided `hadoop-streaming-2.6.5.jar` on the master node and copy it to the `$HADOOP_HOME` folder.

4. Upload all the provided Python scripts in `MR-BWA` folder to the master node

5. Copy the uploaded Python scripts to `app_mrbwa` folder on the master node. This is needed because Python import modules work with symlink, hence files need to reside in a common folder[1]

6. Create `/data/index` folder on master node and download human reference genome in this folder by running the command

```
wget http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips/hg38.fa.gz
```

7. Uncompress the downloaded human reference genome and index it by running the commands

```
gunzip hg38.fa.gz
bwa index -a bwtsw hg38.fa
```

8. Once the index creation has finished, all the BWA index files are copied to `/data/index` folder on all data nodes

9. Download and uncompress the evaluation datasets

[1] `http://stackoverflow.com/questions/18150208/how-to-import-a-custom-module-in-a%2Dmapreduce%2Djob`

```
wget ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/HG00096/
sequence_read/SRR062634_1.filt.fastq.gz
gunzip SRR062634_1.filt.fastq.gz

wget ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3/data/NA12750/
sequence_read/ERR000589_1.filt.fastq.gz
gunzip ERR000589_1.filt.fastq.gz
```

10. Preprocess the `FASTQ` files into the custom MapReduce format by running the provided script

    ```
    fq_to_mrfastq.py <input_file.fastq>
    ```

    The output of this command is a text file called `output.mr.fastq`

11. Upload the custom preprocessed `FASTQ` files to HDFS by running the commands

    ```
    hdfs dfs -mkdir -p /user/karl/mrbwa/output
    hdfs dfs -put output.mr.fastq /user/karl/mrbwa/
    ```

12. Run MapReduce job using the command

    ```
    yarn jar $HADOOP_HOME/hadoop-streaming-2.6.5.jar
    -files app_mrbwa
    -mapper "app_mrbwa/mapper_pybwa.py"
    -combiner "app_mrbwa/combiner_pybwa.py"
    -reducer "app_mrbwa/reducer_pybwa.py"
    -input hdfs:///user/karl/mrbwa/output.mr.fastq
    -output hdfs:///user/karl/mrbwa/output/alignment
    ```

13. Copy tsv output from HDFS to master node using command

    ```
    hdfs dfs -copyToLocal /user/karl/mrbwa/output/alignment/part-00000 .
    ```

B. Setting up MR-BWT-FM on Hadoop

This appendix provides a guide to setup and run `MR-BWT-FM`. It is assumed that an Apache Hadoop Cluster has already been setup as described in Section 4.1 and steps 1, 2, 3 from Appendix A have already been done.

1. Upload all the provided Python scripts in `MR-BWT-FM` folder to the master node

2. Copy the uploaded Python scripts to `app_mrbwtfm` folder on the master node.

3. Assuming that human reference genome has already been downloaded in `/data/index` folder on master node, the custom FM-Index is created by running

   ```
   build_index.py <input_file> hg38_idx
   ```

4. Once the index creation has finished, the `hg38_idx` index file is copied to `/data/index` folder on all data nodes

5. Preprocess the `FASTQ` files downloaded in step 9 of Appendix A, into the custom MapReduce format by running the provided script

   ```
   parse_fq_file.py <input_file.fastq>
   ```

The output of this command is a text file called `output.fq.reads`

6. Upload the custom preprocessed `FASTQ` files to HDFS by running the command

```
hdfs dfs -mkdir -p /user/karl/mrbwtfm/output
hdfs dfs -put output.fq.reads /user/karl/mrbwtfm/
```

7. Run MapReduce job using the command

```
yarn jar $HADOOP_HOME/hadoop-streaming-2.6.5.jar
-files app_mrbwtfm
-mapper "app_mrbwtfm/mapper_native.py"
-combiner "app_mrbwtfm/combiner_native.py"
-reducer "app_mrbwtfm/reducer_native.py"
-input hdfs:///user/karl/mrbwtfm/output.mr.reads
-output hdfs:///user/karl/mrbwtfm/output/alignment
```

8. Copy tsv output from HDFS to master node using command

```
hdfs dfs -copyToLocal /user/karl/mrbwtfm/output/alignment/part-00000 .
```

C. Output Analysis Script

This appendix provides an overview of the `output_analysis.R` that is used to analyze the output explained in Section 3.2.1. This script does not need a Hadoop cluster and is executed on a standard laptop or desktop computer. This script is written using `R` language, which is a free software used for statistical computing. Hence, the machine used to run this analysis script should have `R` installed[1]. Running this script is straightforward:

1. Copy MapReduce output TSV file to some location on the laptop. `output_analysis.R` assumes that output TSV file is called `mrbwa_output.tsv`, however this can be easily changed

2. Open laptop terminal and type `R`. `R` terminal should load up.

3. Run the command

   ```
   source('output_analysis.R')
   ```

4. This will take some time to finish. When it's done, it will create four bar charts based on the input TSV file:

   - `alignment_dist`: This shows the alignment operations distribution. The result shows base pairs count of matches, variations, insertions, deletions.

[1] `https://cran.r-project.org/doc/manuals/R-admin.html`

- `variations_per_chromosome`: This shows the variations base pairs count grouped by chromosome.
- `insertions_per_chromosome`: This shows the insertions base pairs count grouped by chromosome.
- `deletions_per_chromosome`: This shows the deletions base pairs count grouped by chromosome.

D. CD Contents

The attached CD contains the following:

- Soft copy of this report in pdf format
- Source code for `MR-BWA` in a folder called `MR-BWA`
- Source code for `MR-BWT-FM` in a folder called `MR-BWT-FM`
- Output analysis script: `output_analysis.R`
- Hadoop streaming jar file: `hadoop-streaming-2.6.5.jar`
- Installation and setup scripts for third party software: `setup_biotools.sh`, `hd_biotools.sh`

References

[1] U.S. National Library of Medicine, 2016. What is DNA? URL=https://ghr.nlm.nih.gov/primer/basics/dna

[2] Liu, L., Li, Y., Li, S., Hu, N., He, Y., Pong, R., Lin, D., Lu, L., Law, M. 2012. Comparison of Next-Generation Sequencing Systems. Hindawi Publishing Corporation Journal of Biomedicine and Biotechnology Volume 2012, Article ID 251364, 11 pages. URL=http://dx.doi.org/10.1155/2012/251364

[3] Searls, D. 1998. Grand Challenges in Computational Biology. Computational Methods in Molecular Biology, S. Salzberg, D. Searls, and Simon Kasif, eds., Elsevier Science.

[4] Abuin, J. M., Pichel, J.C., Pena, T. F., Amigo, J. 2015. BigBWA: approaching the Burrows-Wheeler aligner to Big Data technologies. Oxford University Press. Bioinformatics 31(24), pp. 4003-4005. URL=https://www.ncbi.nlm.nih.gov/pubmed/26323715

[5] Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., Robinson, G. E. 2015. Big Data: Astronomical or Genomical?. PLoS Biol 13(7): e1002195. doi:10.1371/journal.pbio.1002195

[6] Chen, M. 2015. Next Generation Sequencing. Happy Bioinformatics. URL=http://bio.cwchen.tw/ngs/2015/03/29/next-generation-sequencing/

[7] Motahari, A., Bresler, G., Tse, D. 2013. Information Theory of DNA Shotgun Sequencing. Department of Electrical Engineering and Computer Sciences University of California, Berkeley. arXiv:1203.6233. URL=https://arxiv.org/pdf/1203.6233v4.pdf

[8] Al-Absi, A. A. and Kang, D. K. 2015. Long Read Alignment with Parallel MapReduce Cloud Platform. Hindawi Publishing Corporation BioMed Research International Volume 2015, Article ID 807407, 13 pages. URL=http://dx.doi.org/10.1155/2015/807407

[9] Li, H. and Durbin, R. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics, vol. 25, no. 14, 1754-1760.

[10] Smith, T. F. and Waterman, M. S. 1981. Identification of common molecular subsequences. Journal of Molecular Biology, vol. 147, no. 1, 195-197.

[11] Gotoh, O. 1982. An improved algorithm for matching biological sequences. Journal of Molecular Biology, vol. 162, no. 3, 705-708.

[12] Dean, J. and Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 04), 137-150, San Francisco, Calif, USA.

[13] Lesk, A. M. 2014. Introduction to Bioinformatics. Oxford University Press.

[14] Lodish, H., Berk, A., Zipursky, S. L., Matsudaira, P., Baltimore, D., Darnell, J. 2000. Molecular Cell Biology. 4th edition. New York: W. H. Freeman. Section 1.2, The Molecules of Life. URL=https://www.ncbi.nlm.nih.gov/books/NBK21473/

[15] Sanger, F., Air, G. M., Barrell, B. G., Brown, N. L., Coulson, A. R., Fiddes, C. A., Hutchison, C. A., Slocombe, P. M., Smith, M. 1977. Nucleotide sequence of bacteriophage φX174 DNA. Nature Feb 24, vol. 265, no. 5596, 687-695. DOI=http://dx.doi.org/10.1038/265687a0

[16] Collins, F. S., Morgan, M., Patrinos, A. 2003. The Human Genome Project: lessons from large-scale biology. Science, vol. 300, no. 5617, 286-290.

[17] Robinson R. J. 2014. How big is the human genome?. Precision Medicine.

[18] EMBL-European Bioinformatics Institute 2014. EMBL-EBI annual scientific report 2013.

[19] National Human Genome Research Institute. 2016. The Cost of Sequencing a Human Genome. URL=https://www.gnome.gov/sequencingcosts/

[20] Marx, V. 2013. Biology: The big challenges of big data. Nature, vol. 498, no. 7453, 255-260.

[21] Calabrese, B. and Cannataro, M. 2015. Cloud Computing in Healthcare and Biomedicine. Scalable Computing: Practice and Experience, vol. 16, no. 1.

[22] Langmead B., Schatz, M. C., Lin, J., Pop, M., Salzberg, S. L. 2009. Searching for SNPs with cloud computing. Genome Biol, 2009. 10(11): R134. doi:10.1186/gb-2009-10-11-r134 PMID: 19930550

[23] The International Human Genome Sequencing Consortium 2004. Finishing the euchromatic sequence of the human genome. Nature 431, 931-945.

[24] Langmead, B. and Pritt, J. 2016. Algorithms for DNA Sequencing. Johns Hopkins University. URL=https://www.coursera.org/learn/dna-sequencing

[25] vlab.amrita.edu,. 2012. Global alignment of two sequences - Needleman-Wunsch Algorithm. URL=vlab.amrita.edu/?sub=3&brch=274&sim=1431&cnt=1

[26] Wang, K., Li, M., Hadley, D., Liu, R., Glessner, J., Grant, S. F., Hakonarson, H., Bucan, M. 2007. PennCNV: an integrated hidden Markov model designed for high-resolution copy number variation detection in whole-genome SNP genotyping data. Genome Research vol. 17, 1665-1674.

[27] Baker, M. 2012. De novo genome assembly: what every biologist should know. Nature Methods vol. 9, 333-337. doi:10.1038/nmeth.1935. URL=http://dx.doi.org/10.1038/nmeth.1935

[28] The International SNP Map Working Group 2001. A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. Nature 409, 928-933. DOI=http://dx.doi.org/10.1038/35057149

[29] Sudha Sadasivam, G. and Baktavatchalam, G., 2010. A novel approach to Multiple Sequence Alignment using hadoop data grids. Int. J. Bioinformatics Res. Appl. 6, 5 (January 2010), 472-483. DOI=http://dx.doi.org/10.1504/IJBRA.2010.037987

[30] Srinivasa, K.G., Siddesh, M., Srinidhi, H. 2015. Driving Big Data with Hadoop Technologies. Handbook of Research on Cloud Infrastructures for Big Data Analytics. doi:10.4018/978-1-4666-5864-6.ch010.

[31] Boyer, R.S. and Moore, J.S. 1977. A fast string searching algorithm. Communications of the ACM. 20:762-772.

[32] Gusfield, D. 1997. Algorithms on Strings, Trees and Sequences. University of California, Davis. Cambridge University Press

[33] Simha, R. Module 5, Pattern Search. School of Engineering and Applied Science, The George Washington University. URL=https://www.seas.gwu.edu/~simhaweb/alg/lectures/module5/module5.html

[34] Ukkonen, E. 1995. On-line construction of suffix trees. Algorithmica. 14 (3): 249-260. doi:10.1007/BF01206331

[35] Manber, U. and Myers, G. W., 1993. Suffix arrays: a new method for on-line string searches. SIAM Journal of Computing 22, 5, 935-948.

[36] Berger, B., Peng, J., Singh, M. 2013. Computational solutions for omics data. Nature Reviews Genetics 14, 333-346 doi:10.1038/nrg3433

[37] Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on, pages 390-398.

[38] Peters, T., 2011. [Python-Dev] Sorting. Python Developers Mailinglist. URL=https://mail.python.org/pipermail/python-dev/2002-July/026837.html

[39] Langmead, B. 2016. Dynamic programming and edit distance. Johns Hopkins University. URL=http://www.cs.jhu.edu/~langmea/resources/lecture_notes/dp_and_edit_dist.pdf

[40] Jurafsky, D. and Manning, C. 2014. Natural Language Processing - Minimum Edit Distance. Stanford NLP Group. URL=https://web.stanford.edu/~jurafsky/NLPCourseraSlides.html

[41] Needleman, S. B. and Wunsch, C. D. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology. 48 (3): 443-53. doi:10.1016/0022-2836(70)90057-4

[42] Snyder, W. 2017. CS 112 - Introduction to CS II - Summer II, 2017. Department of Computer Science Boston University. URL=http://www.cs.bu.edu/courses/cs112/Lab09.html

[43] Kashyap, H., Ahmed, H. A., Hoque, N., Roy, S., Bhattacharyya, D. K. 2014. Big Data Analytics in Bioinformatics: A Machine Learning Perspective. Journal of Latex Class Files, vol. 13 no. 9. URL=http://arxiv.org/abs/1506.05101

[44] The 1000 Genomes Project Consortium 2010. A map of human genome variation from population-scale sequencing. Nature 467, 1061-1073. DOI=http://dx.doi.org/10.1038/nature09534

[45] 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., Daly, M. J. 2011. A framework for variation discovery and genotyping using next-generation DNA sequencing data. Nature Genetics vol. 43, 491-498.

[46] Merelli, I., Snchez, H. P., Gesing, S., D'Agostino, D. 2014. Managing, Analysing, and Integrating Big Data in Medical Bioinformatics: Open Problems and Future Perspectives. Hindawi Publishing Corporation BioMed Research International Volume 2014, Article ID 134023.

[47] Langmead, B. and Salzberg, S. L. 2012. Fast gapped-read alignment with Bowtie 2. Nature Methods, vol. 9, no. 4, 357-359.

[48] Pearson W. R. and Lipman, D. J. 1988. Improved tools for biological sequence comparison. Proceedings of the National Academy of Sciences of the United States of America, vol. 85, no. 8, 2444-2448.

[49] Altschul, S. F., Madden, T. L., Schaffer A. A., Zhang, J., Zhang, Z., Miller, W., Lipman, D. J. 1997. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research, vol. 25, no. 17, 3389-3402.

[50] Kent, W. J. 2002. BLAT–the BLAST-like alignment tool. Genome Research, vol. 12, no. 4, 656-664.

[51] Li, R., Li, Y., Kristiansen, K., Wang, J. 2008. SOAP: short oligonucleotide alignment program. Bioinformatics, vol. 24, no. 5, 713-714.

[52] Li, H. and Durbin, R. 2010. Fast and accurate long-read alignment with Burrows-Wheeler transform. Bioinformatics, vol. 26, no. 5, 589-595.

[53] Liu, Y. and Schmidt, B. 2012. Long read alignment based on maximal exact match seeds. Bioinformatics, vol. 28, no. 18, i318-i324

[54] Li, H. 2013. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. URL=http://arxiv.org/abs/1303.3997v1.

[55] Burrows, M. and Wheeler, D. J. 1994. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation

[56] Peters, D., Luo, X., Qiu, K., Liang, P. 2012. Speeding up large-scale next generation sequencing data analysis with pBWA. J. Appl. Bioinform. Comput. Biol., 1, 1-6.

[57] Pireddu, L., Leo, S., Zanetti, G. 2011. SEAL: a distributed short read mapping and duplicate removal tool. Bioinformatics, 27(15), 21592160. DOI=http://doi.org/10.1093/bioinformatics/btr325

[58] Leo, S. and Zanetti, G. 2010. Pydoop: a Python MapReduce and HDFS API for Hadoop. In: Proceeding of 19th Symposium on HPDC, ACM, Chicago (USA), 819-825.

[59] Lin, Y., Yu, C., Lin, 2013. Enabling Large-Scale Biomedical Analysis in the Cloud Hindawi Publishing Corporation BioMed Research International Volume 2013, Article ID 185679.

[60] Lee, D., Hormozdiari, F., Xin, H., Hach, F., Mutlu, O., Alkan, C. 2015. Fast and accurate mapping of Complete Genomics reads. Methods. 2015 Jun;79-80:3-10. doi: 10.1016/j.ymeth.204.10.012. URL=https://www.ncbi.nlm.nih.gov/pubmed/25461772

[61] Hamming, Richard W. 1950. Error detecting and error correcting codes. Bell System Technical Journal, 29 (2): 147-160, doi:10.1002/j.1538-7305.1950.tb00463.x

[62] Levenshtein, V. 1966. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics. Doklady 10 (8) 707-710.

[63] Gonzalez-Dominguez, J., Liu, Y., Schmidt, B. 2016. Parallel and Scalable Short-Read Alignment on Multi-Core Clusters Using UPC++. PLoS ONE 11 (1): e0145490. doi:10.1371/journal.pone.0145490

[64] Georganas, E., Bulu, A., Chapman, J., Oliker, L., Rokhsar, D., Yelick, K. 2015. merAligner: A Fully Parallel Sequence Aligner. In: Proc. 29th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS'15). Hyderabad, India. DOI=http://doi.org/10.1109/IPDPS.2015.96

[65] Liu, Y., Popp, B., Schmidt, B. 2014. CUSHAW3: Sensitive and Accurate Base-Space and Color-Space Short- Read Alignment with Hybrid Seeding. PLOS ONE. 2014; 9(1).

[66] Venkatachalam, B. 2012. Parallelizing the Smith-Waterman Local Alignment Algorithm using CUDA.

[67] Schatz, M. C., 2009. CloudBurst: highly sensitive read mapping with MapReduce. Bioinformatics 2009, 25(11):1363-1369. DOI=https://doi.org/10.1093/bioinformatics/btp236

[68] Nguyen, T., Shi, W., Ruden, D., 2011. CloudAligner: A fast and full-featured MapReduce based tool for sequence mapping. BMC Res Notes, 2011 Jun 6;4:171. doi: 10.1186/1756-0500-4-171

[69] Matsunaga, A., Tsugawa, M., Fortes, J. 2008. CloudBLAST: Combining MapReduce and virtualization on distributed resources for bioinformatics applications. In Proceedings of the 2008 Fourth IEEE International Conference on eScience (ESCIENCE '08). IEEE Computer Society, Washington, DC, USA, 222-229. DOI=http://dx.doi.org/10.1109/eScience.2008.62

[70] Algur, S. P. and Sakri, L. I., 2015. Parallelized Genomic Sequencing Model: A Big data approach for Bioinformatics application. International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), Davangere, 2015, pp. 69-74. doi: 10.1109/ICATCCT.2015.7456857

[71] Algur, S. P. and Sakri, L. I., 2016. An Efficient Bulk Synchronous Parallelized Scheduler for Bioinformatics Application on Public Cloud. International Journal of Computer Applications 145(15):22-30. doi: 10.5120/ijca2016910882

[72] O'Driscoll, A., Belogrudov, V., Carroll, J., Kropp, K., Walsh, P., Ghazal, P., Sleator, R. D. 2015. HBLAST: Parallelised sequence similarity - A Hadoop MapReducable basic local alignment search tool. Journal of biomedical informatics, vol 54, pp. 58-64. DOI=http://dx.doi.org/10.1016/j.jbi.2015.01.008

[73] Meng, Z., Li, J., Zhou, Y., Liu, Q., Liu, Y., Cao, W. 2011. bCloudBLAST: An efficient mapreduce program for bioinformatics applications. 2011 4th International Conference on Biomedical Engineering and Informatics (BMEI), 4, 2072-2076. doi: 10.1109/BMEI.2011.6098717

[74] Cock, P. J. A., Fields, C. J., Goto, N., Heuer, M. L., Rice, P. M. 2010. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. Nucleic Acids Research, 38(6), 17671771. DOI=http://doi.org/10.1093/nar/gkp1137

[75] De Wit, P., Pespeni, M. H., Ladner, J. T., Barshis, D. J., Seneca, F., Jaris, H., Therkildsen, N. O., Morikawa, M., Palumbi, S. R. 2012. The simple fool's guide to population genomics via RNA-Seq: an introduction to high-throughput sequencing data analysis. Molecular Ecology Resources 12, 1058-1067.

[76] Tang, D., 2014. Understanding the BAM flags. Dave Tang's Blog. Computational Biology and Genomics. URL=https://davetang.org/muse/2014/03/06/understanding-bam-flags/

[77] Li, H., Handsaker, B., Wysoker, A., Fennell, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., Durbin, R.; 1000 Genome Project Data Processing Subgroup. 2009. The Sequence alignment/map (SAM) format and SAMtools. Bioinformatics, 25, 2078-9. PMID: 19505943

[78] Vladu, A. and Negruseri, C., 2005. Suffix arrays - a programming contest approach. URL=http://web.stanford.edu/class/cs97si/suffix-array.pdf

[79] Langmead, B., 2013. Introduction to the Burrows-Wheeler Transform and FM Index. Johns Hopkins University. URL=http://www.cs.jhu.edu/~langmea/resources/bwt_fm.pdf

[80] Trelles, O., 2001. On the Parallelization of Bioinformatic Applications. Computer Architecture Department, University of Malaga.

[81] Nahar, S. and Tseng, T. 2016. PSFFX: Distributed Suffix Array Construction. URL=http://snnynhr.github.io/ParallelSuffixArrays/