University of Dublin

TRINITY COLLEGE

Parallel DNA Read Alignment Using the Amazon Cloud

Kayleigh McGinley
Supervisor: Jeremy Jones

Master in Computer Science

School of Computer Science and Statistics
O'Reilly Institute, Trinity College, Dublin 2, Ireland

Submitted to the University of Dublin, Trinity College, April 2019

Declaration

I, Kayleigh McGinley, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

______________________________ ______________

Signed Date

Summary

This dissertation aims to speed up an existing read alignment program thus helping to solve the problem of DNA analysis. The bottleneck of most DNA analysis pipelines is read alignment which is the mapping of short sequences of DNA to a complete sequence of a species' genetic information called a reference genome. The problem with these types of programs is that they have very long run times due to the large amounts of genetic data to be analysed. This dissertation introduces AWS-BWBBLE, an implementation of the existing program BWBBLE by Huang and Popic. It aims to optimize the program by distributing it on the Amazon cloud.

A research question is posed asking if a straightforward approach exists that will give BWBBLE a linear speed-up when its work is distributed amongst a number of virtual machines on the cloud. There is a list of objectives presented which need to be accomplished in order to answer the posed question.

As a dissertation for The School of Computer Science and Statistics in Trinity College Dublin, the area of bioinformatics will be introduced, and its biology influences will be explored in detail. This focusses at the areas of genetics and genomics which are central to the bioinformatics field. Next-generation DNA sequencing methods are discussed as well as the speeds they are reaching. The theoretical background of read alignment is explained with focus on the Burrows-Wheeler Transform and its use in read alignment programs.

Other read alignment programs are investigated and compared to the BWBBLE program. BWBBLE is widely explored both theoretically and practically, with the code and its uses discussed. Previous work to parallelize alignment programs are presented and their results analysed, including a previous attempt to parallelize the BWBBLE program using Apache Spark on the Google cloud. Cloud computing and distributed systems are explained and the benefit of them on this program is discusses.

The design of the program is presented with emphasis on the proposed architecture for the distributed system. The popularity of AWS is highlighted and several of their services are explored more closely. Amazon's virtual machine instances and elastic file storage options are investigated, and pricing is presented for all decisions made.

Docker is introduced, and its growing world-wide interest explored. Containerization is compared with the standard virtual machine. Docker is explored in greater detail with focus on Docker containers and images. A method to share files between a container and its host machine is investigated.

The implementation of the project is then discussed in detail starting with the modification of the BWBBLE code by adding a new set of optional parameters. The method of setting up the AWS services through the AWS Management Console is then explained, including the configuration of virtual machines and allowing network access to the shared file system. Finally, the automation of the system using a shell script with Amazon CLI commands is explained in great detail.

The results are presented in three sections; the initial results, testing of the built-in multithreading and lastly, comparing the results to previous work. The results are separated as such due to encountering a problem with the original BWBBLE code which is also discussed in this section.

The conclusion finds the project to have been a success as the methodology was kept straightforward while also achieving a linear speed-up using up to 4 VMs.

// Abstract

Parallel DNA Read Alignment Using the Amazon Cloud

By Kayleigh McGinley | Supervised by Jeremy Jones

Trinity College Dublin | School of Computer Science & Statistics

Master in Computer Science

Short-read alignment is the process of searching for short sequences of DNA within a species' entire set of genes. Many short-read alignment programs exist, such as BWA[1], SOAP2[2] and Bowtie[3]. However, these programs all have one thing in common; they use a single reference genome. The use of a single reference genome can lead to inherent biases and lower accuracy. BWBBLE was created to map short reads to a collection of genomes (a reference multi-genome) with high accuracy. It handles genetic variants thus avoiding the inherent bias to one specific genome[4].

One major concern with BWBBLE is that it is up to 100 times slower than other read aligners due to the larger amount of data it processes. The aim of this project is to introduce a new version of BWBBLE, called AWS-BWBBLE that uses Amazon Web Services to distribute the work amongst a number of virtual machines. A small distributed network was successfully created in AWS using Elastic Compute Cloud instances (VMs) and Elastic File Storage. The parallelization is achieved by instructing each VM in the network to process a different subset of the reads file. This straightforward approach was a complete success as proved by the linear speed-up of the program using up to four VMs.

---

[1] LANGMEAD, B., TRAPNELL, C., POP, M. & SALZBERG, S. L. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome biology, 10.
[2] LI, H. & DURBIN, R. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics (Oxford, England), 25.
[3] LI, R., YU, C., LI, Y., LAM, T.-W., YIU, S.-M., KRISTIANSEN, K. & WANG, J. 2009. SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics (Oxford, England), 25.
[4] HUANG, L. & POPIC, V. 2013. Short read alignment with populations of genomes. Bioinformatics (Oxford, England), 29.

Acknowledgements

To my supervisor Jeremy Jones, who proposed this project and continuously helped me throughout. Your encouragement and advise is the driving force behind this work.

To my family, for their constant trust and belief in me and my academic capabilities over the past five years.

To my friends, who have supported me in every aspect of my college life and pushed me to achieve my goals both in academics and in sport during my time in Trinity.

Table of Abbreviations

Abbreviation	Full Name	Description
AMI	Amazon Machine Image	Deployable image of the desired state of an EC2 instance
AWS	Amazon Web Services	Amazon’s cloud computing services
BWA	Burrows-Wheeler Alignment	Read alignment package based on backward search with Burrows-Wheeler Transform
BWBBLE	N.A.	BWT-based aligner
BWT	Burrows-Wheeler Transform	A transform that rearranges a character string into runs of similar characters
CLI	Command Line Interface	Provides control of AWS using the command line
CPU	Central Processing Unit	Piece of hardware that carries out the instructions of a computer program
DNA	Deoxyribonucleic acid	A molecule that contains the instructions an organism needs to develop, live and reproduce
DNS	Domain Name System	System to track and regulate internet domain names
EC2	Elastic Computer Cloud	Provides storage, processing, and Web services to customers via AWS
EFS	Elastic File Storage	Provides affordable, scalable storage via AWS
FASTA	N.A.	Text-based format for representing DNA or protein sequences
GCP	Google Cloud Platform	Google’s cloud computing services
ID	Identification/Identity	Used to establish the identity of someone or something
IFS	Internal Field Separator	A shell variable for determine word splitting
IP	Internet Protocol	Unique identifier for a computer
NFS	Network File System	Distributed file system protocol that allows access to files over a network

NGS	Next-Generation Sequencing	Umbrella term to describe modern sequencing technologies
Occ	Occurrence	An incident or event. In this case, refers to the Occ table BWT data structure
PEM	Privacy Enhanced Mail	File format for cryptographic keys
SCP	Secure Copy	Allows copying of files between two locations, e.g. from a local server to a remote server
SSH	Secure Shell	Network protocol for operating network services securely
STDIN	Standard Input	Input from the keyboard
TLS	Transport Layer Security	Cryptographic protocol for end-to-end communication security over networks
vCPU	Virtual Central Processing Unit	CPUs assigned to a virtual machine
VM	Virtual Machine	An emulation of a computer system
VPC	Virtual Private Cloud	Used to isolate desired elements of the AWS

List of Figures

Figure 1 Stylized Diagram of DNA (Russel, 2010) 5
Figure 2 Comparing the Size of The Human Reference Genome and Reads 6
Figure 3 Generating Suffix Array For "GATCGTACC$" 7
Figure 4 Burrows-Wheeler Transform of "GATCGTACC$" 8
Figure 5 BWT "First-Last Property" 8
Figure 6 Backwards Search Example 9
Figure 7 The Occ Table for the BWT "CTGCAT$CGA" 10
Figure 8 Correlation Between C Array and Sorted Suffixes 10
Figure 9 Comparing Run Time of Alignment Program with and Without BWT 11
Figure 10 BWBBLE Read Alignment Options 14
Figure 11 Comparing Speed-Up Times of SparkBWA and BigBWA 15
Figure 12 Speed-Up of SparkBWBBLE 16
Figure 13 Comparing Virtual Machine and Docker Container Architectures 20
Figure 14 Docker File to Image to Container 21
Figure 15 Showing EFS Access From Docker Container 21
Figure 16 BWBBLE Updated Read Alignment Options 22
Figure 17 Sample of reads file 23
Figure 18 EFS File System Access 24
Figure 19 Create AMI in AWS Management Console 27
Figure 20 Depiction of the Splitting of Reads in the Shell Script 33
Figure 21 Speed-Up of AWS-BWBBLE Using 4 Threads 37
Figure 22 Comparing the Speed-Up of The Original BWBBLE Program Using the Multithreaded Option 38
Figure 23 Linking AWS-BWBBLE's Lack of Speed-Up to BWBBLE's Threads 39
Figure 24 Comparing the Speed-Up of AWS-BWBBLE Using BWBBLE's In-Built Multithreaded Option 40
Figure 25 Run Times of Multithreaded AWS-BWBBLE 40
Figure 26 Comparing the Speed-Up of AWS-BWBBLE and SparkBWBBLE 41

List of Tables

Table 1 t2.xlarge instance details 18
Table 2 Run Times of AWS-BWBBLE 52
Table 3 Speed-Up of AWS-BWBBLE 52
Table 4 Speed-Up of Multithreaded BWBBLE 52

Table of Contents

Summary ii

Abstract iv

Acknowledgements v

Table of Abbreviations vi

List of Figures viii

List of Tables ix

1. Introduction 1

1.1. Motivation 1

1.2. Research Question 2

1.3. Research Aims 3

1.4. Research Objectives 3

1.5. Overview of This Report 4

2. Background 5

2.1. Genetics and Genomics 5

2.1.1. DNA and Nucleotides 5

2.1.2. Reference Genomes 5

2.2. DNA Sequencing and Analysis 6

2.2.1. Sequencing and Sequenced Reads 6

2.2.2. Read Alignment 6

2.3. Burrows-Wheeler Transform 7

2.3.1. The Suffix Array 7

2.3.2. The Transform 8

2.3.3. Backwards Search 9

2.3.4. The Occ Table 10

2.3.5. Use in Read Alignment 11

3. State of the Art 12

3.1. Single Genome Read Alignment Programs 12
3.1.1. Burrows-Wheeler Alignment 12
3.1.2. SOAP2 12
3.1.3. Bowtie 12
3.2. BWBBLE 13
3.2.1. The BWBBLE Code 13
3.3. Parallelized Programs 15
3.3.1. SparkBWA 15
3.3.2. SparkBWBBLE 16
4. Design 17
4.1. Cloud Computing 17
4.1.1. Distributed Cloud Computing 17
4.2. Amazon Web Services 18
4.2.1. Elastic Compute Cloud 18
4.2.2. Elastic File Storage 19
4.2.3. Command Line Interface 19
4.3. Docker 20
4.3.1. Comparison of Docker Containers and Virtual Machines 20
4.3.2. Docker Images 21
4.3.3. Docker Bind Mount 21
5. Implementation 22
5.1. Generation of Test Files 22
5.2. Code Modification 22
5.3. AWS Setup 24
5.3.1. Elastic File Storage 24
5.3.2. Amazon Machine Image 24
5.4. Parallelizing BWBBLE 28

5.4.1. Allocating CPUs in Docker and EC2 28
5.4.2. Multithreaded Version of BWBBLE 28
5.5. Automation 29
5.5.1. CLI Configuration 29
5.5.2. Shell Script 30
6. Results 37
6.1. Initial Testing 37
6.2. Testing Built-In Multithreading 38
6.3. Testing Different Numbers of Threads 40
6.4. Comparing Results to Previous Work 41
7. Conclusion 42
7.1. Research Question and Objectives 42
7.2. Overview of the Results 43
7.3. Final Remarks 43
8. Bibliography 44
9. Appendices 46
9.1. Appendix 1: Setting BWBBLE Options Based on Alignment Parameters 46
9.2. Appendix 2: Function to Load The Read Sequences from The FASTQ File 47
9.1. Appendix 3: Sequential and Parallel Inexact Matching Functions 49
9.2. Appendix 4: Results Data Tables 52

1. Introduction

This chapter is an introduction to the project, presenting its motivations and aims. It outlines the research question that this body of work will attempt to answer as well as the objectives that the project must achieve to accurately answer the proposed question.

1.1. Motivation

This dissertation focusses on parallelizing an existing read alignment program using the Amazon cloud. The project lies within the area of bioinformatics which uses computer science to help solve complicated biological problems. Short-read alignment is a common first step during genomic data analysis and plays a critical role in medical and population genetics (Huang and Popic, 2013). The term describes the process of locating reads within a large reference genome. Reads are short sequences of DNA and the reference genome is a representation of the entire set of genes for a species.

The speed of next-generation sequencing is expected to increase 3 to 5 times each year. This equipment is responsible for generating the reads that are then mapped using alignment tools. Current alignment tools are not capable of dealing with this growing amount of data, some even taking days to process the data (Arram et al., 2017). Read alignment is complex because both exact and inexact matches may occur during the alignment process. Since sequenced data is too complex for the human-eye to accurately analyse there is a growing strain on the computing systems. Short-read alignment is the bottleneck of most of these sequence analysis pipelines, making it the most obvious target for improvement (Arram et al., 2017).

In 2015 an international research project announced that it had successfully sequenced 1000 human genomes. The project set out to provide a comprehensive description of human genetic variation by sequencing the genomes of a diverse set of 1000 individuals (The Genomes Project, 2015). Most read aligners are run against a single genome which means that they are not taking the genetic variants that were found during this project into account. This project will explore BWBBLE which was created in 2013 while the 1000 Genome Project was in progress. This program accounts for genetic variations by processing a reference multi-genome opposed to a singular genome. Although BWBBLE has much slower run times to some other alignment tools, it is found to be a much more efficient solution than running a single-aligner multiple times on a collection of genomes (Huang and Popic, 2013).

There is a need for alignment tools to achieve greater speeds to keep up with the growing amount of sequenced data. This is a common issue in bioinformatics where computational approaches are needed to solve a problem. A lot of processing power will be required to speed up read alignment, so this research will investigate utilising cloud computing. Cloud computing gives access to remote servers over the internet and can be used for storage and computing power. Amazon Web Services was chosen as the cloud computing provider for this project due to its overall command of the cloud computing market and its existing use within bioinformatics.

1.2. Research Question

The main intention of this research is to answer the following question:

"Does a straightforward approach exist that will give the BWBBLE DNA analysis program a linear speed-up when it's work is distributed amongst a number of virtual machines on the Amazon Cloud?"

The project will take the existing BWBBLE program and attempt to distribute the work amongst a number of virtual machines on the Amazon cloud. The research question highlights that the project will keep the design and implementation as straightforward as possible. The project anticipates that for each machine added to the cloud cluster, the run-time should decrease in a linear motion.

1.3. Research Aims

BWBBLE is a bioinformatics program using graph-based read alignment to analyse DNA. It is written in the C programming language and is used to locate the positions of many millions of short DNA read sequences in a reference genome. Since the human reference genome contains approximately 3 billion base pairs (Human Genome Sequencing Consortium, 2004) it can take a long time for the alignment process to complete. The aim of this dissertation is to see if it is possible to attain a satisfactory speed-up of the BWBBLE DNA analysis program by running it on the Amazon Cloud using a straightforward approach.

This project aims to achieve a linear speed-up by parallelizing the read alignment process. This will be done by distributing the work amongst several worker nodes. Each worker node will be given a subset of the reads to align and will produce their own output file of the locations of those reads. The reads and reference genome will be stored on a shared file system to avoid duplicating the data on each machine. Each VM will be instructed as to which section of reads they must process. The output files will be collected when the processes have completed and combined, producing a final output file.

1.4. Research Objectives

To accomplish the stated research question and aims of this dissertation the following computational objectives must be achieved:

1. The BWBBLE code modified to accept start and end points of the reads file as parameters for the alignment.
2. A shared file system set up to be accessed by the VMs in the distributed system.
3. A deployable machine image created with access to the shared file system with the BWBBLE code built and ready to run inside.
4. An automated script created to fully automate the system using the existing machine image and shared file system.
5. The results file downloaded to the host machine for access after the distributed system is shut down.

1.5. Overview of This Report

The remainder of the dissertation is broken down into the following chapters:

Chapter 2 explores the background of the area of genetics and bioinformatics. In particular, it looks at the burrows-wheeler transform that the BWBBLE program is based off. It exists to give the reader a basic understanding of the field and purpose of this research.

Chapter 3 explains relevant read alignment programs and prior work to parallelize and optimize these programs. It also looks at the project that inspired this research.

Chapter 4 describes the design decisions made for the project. It investigates a distributed cloud network is created and displays the research completed on relevant services from AWS. Docker is introduced and discussed in this chapter.

Chapter 5 details the implementation of the proposed design outlined in chapter 4. It describes the cluster preparation and deployment. It explains the set-up of Docker and AWS as well as the distribution of work and the merging of the results files.

Chapter 6 discusses the final results. These results are compared to similar programs to see if the project has been satisfactory. The run time is evaluated to determine if a sufficient speed-up has been achieved.

Chapter 7 holds the final remarks on the project. It reflects on the design, implementation and results of the project and finally concludes whether the project has been a success.

2. Background

The subject area of this dissertation is known as Bioinformatics. This is the interdisciplinary field of Computer Science and Biology. It involves using computers to help solve complex biological questions. Genetics and genomics are common research areas within bioinformatics due to them focussing on DNA and genomic data. This chapter will explore these areas to give the reader a better understanding of the problem at hand.

2.1. Genetics and Genomics

Genetics is the central to biology since all living things have genes. Genes are comprised of DNA and each person's genes are a combination of both their parent's sets of genes. A genome is a complete set of genes. Genomics is the science of obtaining and analysing the sequences of genomes. Complete genomic DNA sequences have been defined for many viruses and organisms, including humans (Russel, 2010). There is a large and rapidly growing amount of sequenced DNA data available (Dale et al., 2012).

2.1.1. DNA and Nucleotides

DNA are large molecules that consist of many smaller molecules called monomers. The monomers in DNA are called nucleotides and consist of a base. Bases are represented by one of the four characters A, T, G or C which stand for adenine, thymine, guanine and cytosine respectfully. As seen in figure 1, these bases pair up to form the DNA double helix. The adenine and thymine bases are always paired together just as the guanine bases always pair with cytosine. The nucleotide sequence is the most detailed information that can be obtained about DNA. When complete sequences are compared, they can tell us how closely related two organisms are (Russel, 2010).

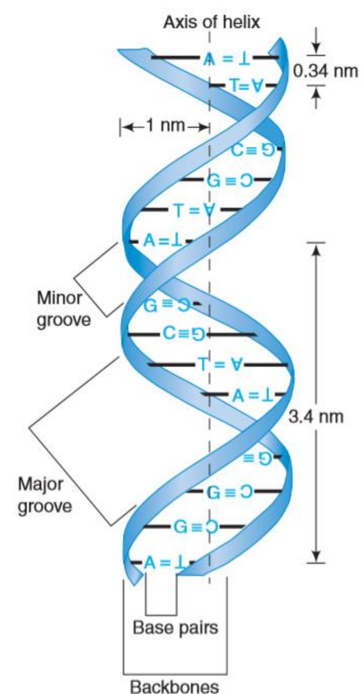

Figure 1 Stylized Diagram of DNA (Russel, 2010)

2.1.2. Reference Genomes

A reference genome is a representation of a species' full set of genes. These reference genomes are assembled by scientists and can take years to sequence. The Human Genome Project was an international scientific research project that started in 1990 and spanned 6 countries and 20 institutions (Lee, 1991). Its goal was to sequence the entire

human genome. The genomes of other well-studied organisms such as, e. coli, the fruit fly and the mouse were also partially sequenced as trial runs during the project (Russel, 2010). The project was complete in 2003 when the group successfully sequenced the human genome. The human reference genome is about 3 billion base pairs in length.

2.2. DNA Sequencing and Analysis

2.2.1. Sequencing and Sequenced Reads

DNA sequencing is the process of determining the order of the nucleotide bases that make up DNA. Next-generation sequencing equipment is used to generate reads and is expected to increase 3 to 5 times in speed each year. The latest NGS platforms are capable of generating terabytes of data in a single run (Arram et al., 2017). This sequenced data is too complex for the human-eye to accurately analyse so computer programs are being created to help with this.

2.2.2. Read Alignment

Read alignment is the process of finding the location of reads within a reference genome. It is essentially a fuzzy search for a substring within a much larger string. Figure 2 gives an idea of the substantial difference in size between the human reference genome and the reads. A common analogy is to imagine the reads as jigsaw puzzle pieces and the reference genome as the image of the completed puzzle on the cover of the box (Stratford, 2018).

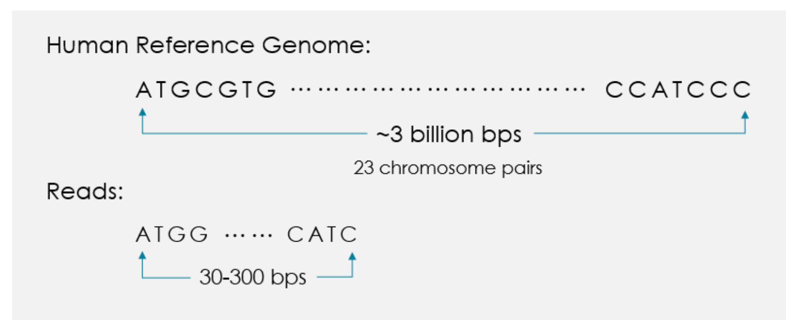

Figure 2 Comparing the Size of The Human Reference Genome and Reads

Short-read alignment is the bottleneck of most sequence analysis programs. Current alignment tools can take days to process the sequenced reads (Arram et al., 2017). Improvement is needed for these alignment tools to keep up with the growing amounts of sequenced data.

2.3. Burrows-Wheeler Transform

The Burrows-Wheeler Transform is a reversable permutation of a string and was initially used as a compression algorithm. It allows large texts to be search efficiently in a small memory footprint (Huang and Popic, 2013). The transform takes a string as input with '$' appended to denote the end of the string. This section will explain how the BWT is created as well as the indexed data structure that is used in real-world applications using a sample string of bases as an example ("GATCGTACC$").

2.3.1. The Suffix Array

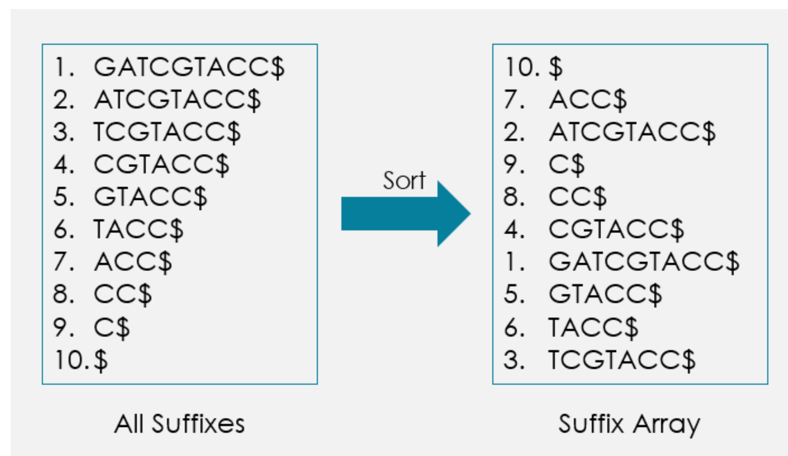

Figure 3 Generating Suffix Array For "GATCGTACC$"

A suffix array is a sorted array of all the suffixes of a string. A suffix is the end of a string from a given position. To create a suffix array, all the suffixes of that string are listed and then sorted alphabetically. Binary search is used to locate a substring using the suffix array. Binary search is a well-known search algorithm for locating the position of an object within a sorted array. It starts by comparing the target value to the element in the middle of the array, it then deciphers which half of the array the value lies within and discards the other half of the array. This method continues until a match is found or in this case, until a suffix is found that begins with or equals the substring we are searching for.

2.3.2. The Transform

	BWT	Sorted Suffixes
10.	C	$
7.	T	ACC$
2.	G	ATCGTACC$
9.	C	C$
8.	A	CC$
4.	T	CGTACC$
1.	$	GATCGTACC$
5.	C	GTACC$
6.	G	TACC$
3.	A	TCGTACC$

BWT = "CTGCAT$CGA"

Figure 4 Burrows-Wheeler Transform of "GATCGTACC$"

The transform is found using the suffix array. The preceding character of each suffix makes up the BWT. The letters in the BWT text hold the same relative positions as in the original text. Meaning that the nth instance of a letter in the BWT is the nth instance of that letter in the original text. Figure 5 demonstrates this more clearly.

	BWT	Sorted Suffixes
10.	C	$
7.	T	ACC$
2.	G	ATCGTACC$
9.	C	C$
8.	A	CC$
4.	T	CGTACC$
1.	$	GATCGTACC$
5.	C	GTACC$
6.	G	TACC$
3.	A	TCGTACC$

Figure 5 BWT "First-Last Property"

As seen above, the first 'C' in the BWT corresponds to the first 'C' in the sorted suffixes. For example, if you take the first instance of 'C' in the BWT and concatenate it with its corresponding suffix ("$") it will equal the first suffix beginning with 'C' ("C$"). The same is highlighted in the figure for the first instance of 'T' and the second instance of 'C'. The same holds for all characters in the BWT. This is known as the "First-Last Property" (Filion, 2016).

2.3.3. Backwards Search

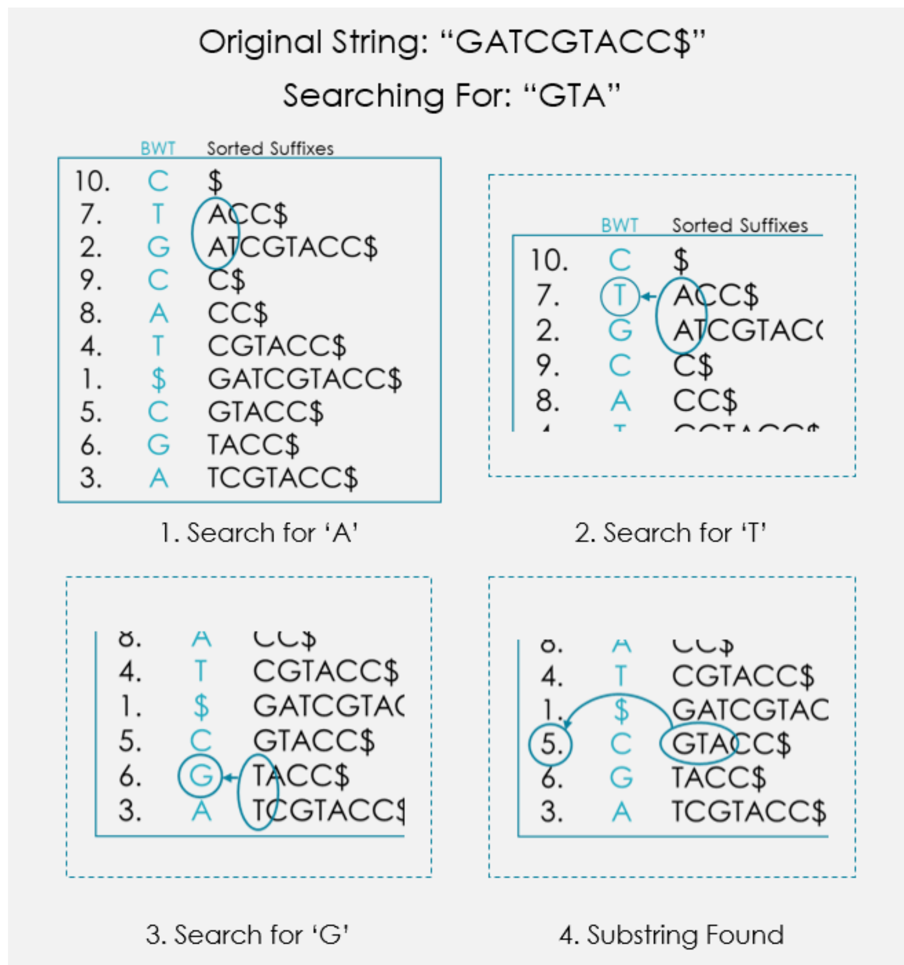

Figure 6 Backwards Search Example

Backwards search is named as such because it begins the search with the last character in the string and works towards the first. Figure 6 shows an example of backwards search looking for the substring "GTA" within the sample string "GATCGTACC$". Since backwards search starts at the end of the string, we first search the suffix array for a suffix that begins with 'A'. All instances of 'A' as the first letter of a suffix are noted. Backwards search makes use of the BWT when searching as it holds the preceding character to the first character of each suffix. As we can see in the figure, the only suffix with 'T' preceding 'A' is that at index 7 meaning that the substring will be at an index below 7. We then continue with our search to find that 'G' is present in the BWT preceding 'T' at index 6. Finally, we see that the substring is present at index 5.

2.3.4. The Occ Table

BWT	A	C	G	T
C	0	1	0	0
T	0	1	0	1
G	0	1	1	1
C	0	2	1	1
A	1	2	1	1
T	1	2	1	2
$	1	2	1	2
C	1	3	1	2
G	1	3	2	2
A	2	3	2	2

OCC

C: $ A C G T

Figure 7 The Occ Table for the BWT "CTGCAT$CGA"

The data structure that is used in real-world applications is called the Occ table. The table contains the cumulative number of occurrences of each character in the BWT (Filion, 2016). As shown in figure 7, there is a column for each nucleotide character in alphabetical order. The numbers correspond to the occurrences of the character at that position. For example, the ‘A’ column remains at 0 until the first occurrence of ‘A’ in the BWT which is the fifth character, it then holds the value of 1 until the next occurrence which is the last character where the value is increased to 2. The Occ table is accompanied by an array called the C array which holds the index of the first occurrence of each character in the sorted text (Filion, 2017).

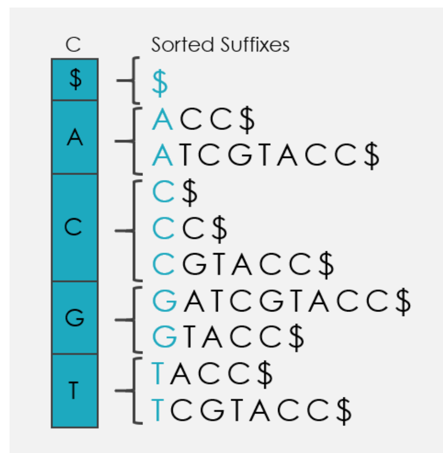

Figure 8 Correlation Between C Array and Sorted Suffixes

The C array corresponds to the first character in the sorted suffixes. It can be said that $C[X]$ is the position of the first X in the suffix array (Filion, 2017). Furthermore, $C[\$]$ will always be 0 and $C[A]$ will always be 1 since there should only ever be one occurrence of ‘$’ in a string.

2.3.5. Use in Read Alignment

The Burrows-Wheeler Transform is commonly used in read alignment programs. It is used transforms a genome into an indexed data structure which brings the run time of programs from about the length of the genome down to about the length of the read. Read alignment programs will usually pre-calculate the BWT of a single reference genome and then map the sequenced reads to it using a variant of the BWT backwards search (Huang and Popic, 2013).

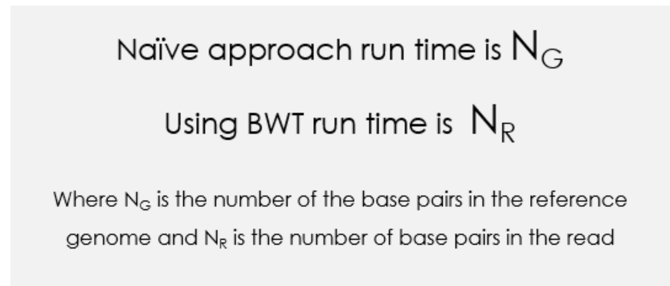

Figure 9 Comparing Run Time of Alignment Program with and Without BWT

3. State of the Art

This chapter discusses some of the existing read alignment programs and the motivations behind their creation. It focusses on the existing state of the BWBBLE program and some of the previous attempts to parallelize the program.

3.1. Single Genome Read Alignment Programs

3.1.1. Burrows-Wheeler Alignment

Burrows-Wheeler Alignment (BWA) is an alignment tool based on backward search with Burrows-Wheeler Transform. The motivation behind BWA was to create a read alignment program that supports gapped alignment for single-end reads. Gapped alignment allows the algorithm to match sequences that have minor differences. Introducing these gaps can allow an alignment algorithm to match more terms than a gap-less algorithm. (Li and Durbin, 2009)

3.1.2. SOAP2

This alignment program was created to replace the preceding short oligonucleotide alignment program (SOAP). SOAP uses excess memory compared to other alignment programs as it loads full reference sequences into memory. It uses a hash look-up table to increase alignment speeds which is also stored in memory. (Li et al., 2008) SOAP2 aimed to both increase alignment speeds and reduce computer memory usage. It introduces a new BWT compression index that was not used in the previous version that reduces the memory usage by indexing the reference sequence in main memory. This caused the memory usage to reduce from 14.7 to 5.4 gigabytes and alignment speeds to improve by 20-30 times. (Li et al., 2009)

3.1.3. Bowtie

Bowtie is known as the ultrafast, memory-efficient alignment program. It’s speed and minimal memory usage is due to its use of the Burrows-Wheeler index along with a backtracking algorithm that allows mismatches. In read alignment mismatching allows for the alignment of a read even if some of the bases do not match. Unlike many other alignment programs, Bowtie creates a permanent index of the reference genome that can be used for future runs of the program. It uses very little memory (1.3 GB) meaning that it can be run on a basic PC with as little as 2 GB of RAM. Bowtie does not support pair-end or gapped alignment. (Langmead et al., 2009)

3.2. BWBBLE

BWBBLE uses graph-based read alignment to analyse DNA. As discussed above, many efficient short-read alignment programs already exist. However, these programs all use a single reference genome which can lead to inherent biases and lower accuracy. BWBBLE was created to map short reads to a collection of genomes (a reference multi-genome) with high accuracy. It handles genetic variants thus avoiding the inherent bias to one specific genome (Huang and Popic, 2013). BWBBLE supports both gapped alignment and mismatching.

One major concern with BWBBLE is that it is very slow in comparison to other aligners. BWBBLE is almost 100 times slower than BWA (Huang and Popic, 2013). This is due to it mapping to multiple reference genomes instead of just one, which is a lot more work for the program. However, to achieve the same result with BWA, it would have to be run multiple times to process each genome in the collection. When this is considered, BWBBLE is a much more efficient method and guarantees much more accuracy than the other programs due to its lack of bias. There is also an optional mode for aligning to a single genome in BWBBLE which is a lot faster but not utilized in this project.

3.2.1. The BWBBLE Code

The program is written in the C programming language and is available on GitHub (Huang and Popic, 2015). The program consists of two subdirectories; the *"mg-ref"* by Huang contains scripts to create the reference multi-genome in the standard FASTA format. A sample reference multi-genome is provided by the project which is used during testing. The second subdirectory is the *"mg-aligner"* which was created by Popic and contains the alignment code of BWBBLE which operates in three steps:

1. Reference Indexing

This is the creation of the indexed data structure using BWT. It is run with the reference genome file as a parameter using this command:

$$bwbble\ index\ < seq_fasta >$$

2. Read Mapping

The BWBBLE read alignment algorithm is based on BWT backwards search to locate reads in the reference multi-genome. BWBBLE expanded upon BWA's inexact search algorithm to allow for mismatches and gaps (Huang and Popic, 2013).

The following command is used to run the alignment of the reads to the reference multi-genome:

$$bwbble\ align\ [options] < seq_fasta >$$
$$< reads_fastq > < aln_results >$$

Where;

- $< seq_fasta >$: The reference multi-genome.
- $< reads_fastq >$: The reads file.
- $< output_aln >$: The specified file to send the output results of the read alignment.
- $[options]$: The possible input parameters to the alignment (see figure 10).

```
Options: M    mismatch penalty (default: 3)
         O    gap open penalty (default: 11)
         E    gap extend penalty (default: 4)
         n    maximum number of differences in the alignment (gaps
              and mismatches) (default: 0)
         l    length of the seed (seed := first seed_length chars
              of the read) (default: 32)
         k    maximum number of differences in the seed (default: 2)
         o    maximum number of gap opens (default: 1)
         e    maximum number of gap extends (default: 6)
         t    run multi-threaded with t threads (default: 1)
         S    align with a single-genome reference
         P    use pre-calculated partial alignment results
```

Figure 10 BWBBLE Read Alignment Options

The figure above shows the in-built options of BWBBLE. This allows the user to change various options when running the alignment command. The most interesting of these options is the in-built multithreaded option. This is implemented using a read counter ($reads -> count$) that is initialized when the reads file is first read and incremented for every read processed. This counter is used to split the reads into batches for each thread.

3. Alignment Results Evaluation & Reporting

This command converts the alignment output to the standard SAM format:

$$bwbble\ aln2sam < seq_fasta > < reads_fastq >$$
$$< aln_results > < output_sam >$$

3.3. Parallelized Programs

3.3.1. SparkBWA

BWA is one of the most widely adopted read alignment tools. A mentioned in section 3.1.1, it is based on backward search with BWT (Li and Durbin, 2009). SparkBWA exploits the capabilities of Apache Spark, a cluster computing framework, to boost the performance of BWA. BWA consists of many algorithms that that were created specially to deal with the alignment of short reads. SparkBWA doesn't modify BWA, but instead adds on the exiting code to maintain compatibility (Abuín et al., 2016).

SparkBWA was designed with the intension to outperform BWA and other BWA-based aligners. Figure 11 compares the speed-up achieved with SparkBWA with that of another BWA-based aligner, BigBWA. BigBWA is a similar program that was previously created by the authors of SparkBWA. However, it uses the Big Data technology Hadoop, a distributed computing framework, to boost the performance of BWA (Abuín et al., 2015). Hadoop is the predecessor to Spark and both offer similar functionalities.

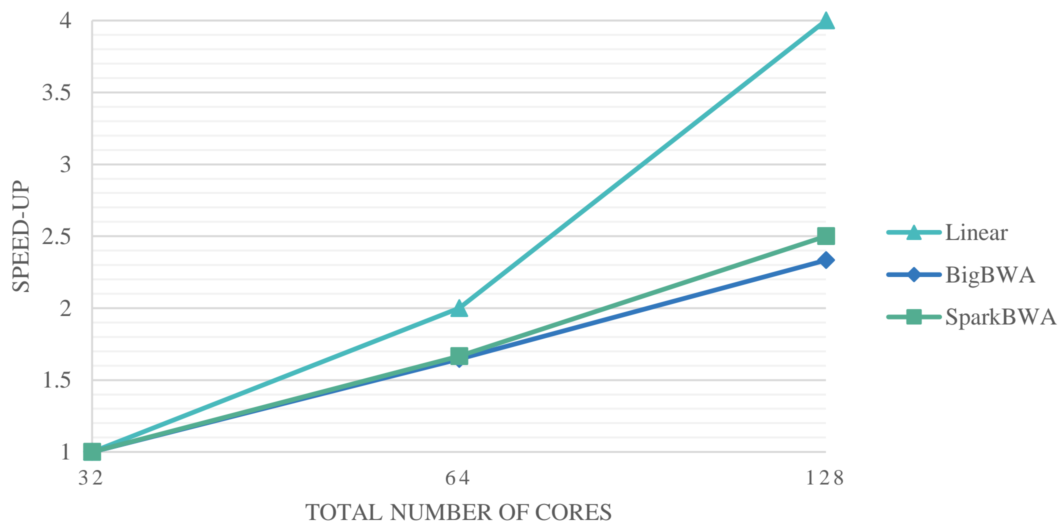

Figure 11 Comparing Speed-Up Times of SparkBWA and BigBWA

The graph above shows the speed-up achieved from both SparkBWA and BigBWA as well as a linear line to show the desired linear speed-up from such a program. It is clear from the graph that SparkBWA achieved some speed-up when compared to its predecessor but neither program is achieving the linear speed-up that is believed to be possible.

3.3.2. SparkBWBBLE

SparkBWBBLE was created for a Bachelor of Computer Science thesis in 2018 by Trinity College Dublin alumnus, Ben Stratford (Stratford, 2018). The project aimed to replicate the design of SparkBWA using the BWBBLE program. During this project, BWBBLE was successfully deployed on a Google Dataproc cluster using Apache Spark. Google Dataproc is a service from the Google Cloud Platform that manages Spark and Hadoop services.

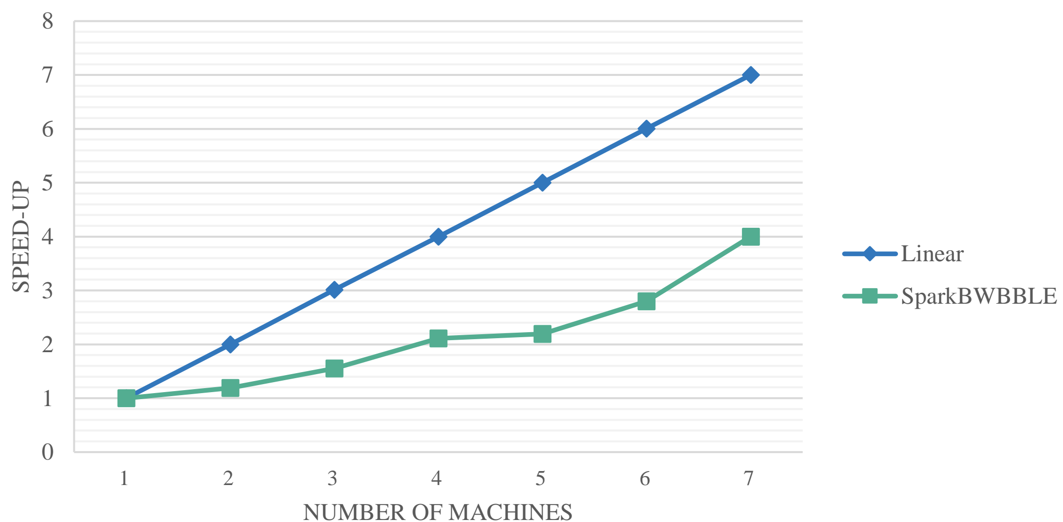

Figure 12 Speed-Up of SparkBWBBLE

The project suggests that the method of using Spark for the parallelization of BWBBLE is not the best approach. This conclusion is drawn from the difficulties encountered throughout the project and the final results not meeting the expectations of the project. Figure 12 shows the speed-up of the SparkBWBBLE program on a growing number of VMs. These results are inadequate when compared to the linear speed-up that the project was aiming to achieve. This project was the inspiration for the straightforward approach described in this dissertation.

4. Design

This chapter explains the design of the distributed system and how it makes use of various services from AWS. It also introduces Docker and its purpose for the project.

4.1. Cloud Computing

Cloud computing aims to give access to large amounts of computing power in a fully virtualized way. For computing to be considered fully virtualized it must allow computers to be built from distributed components such as processing, storage, data and software resources. Cloud computing has become an umbrella term to describe a category of on demand computing services, such as those offered by Amazon (AWS), Google (GCP) and Microsoft (Microsoft Azure) (Buyya et al., 2011).

4.1.1. Distributed Cloud Computing

A distributed system is a collection of independent but interconnected computers that appear as one single coherent system. These systems of multiple computers can achieve the workload of a high-performance supercomputer. Cloud computing systems consist of virtual machines which facilitate distribution and parallelization. All the VMs can be controlled by a single physical machine. Distributed computing on the cloud offers a lot of flexibility (Mahajan and Shah, 2013).

Distributed computing is the method of executing different parts of a program on multiple machines at the same time. Meaning that it requires the program to be separated into executable pieces that can be run concurrently (Rehman, 2018). The splitting of work is important as it ensures the program is making full use of each of the VMs in the network.

4.1.1.1. Distribution of Reads

This project deals with an embarrassingly parallel problem. Since the reads file is processed sequentially from start to end, there will be absolutely no overlap in the work if the reads file is split and distributed to different VMs. There are two possible methods to do this; physically split the file and generate a number of smaller files to be sent to the VMs, or, send location pointers to each VM to identify which portion of the reads file to process. As I will later discuss, the project made use of a shared file system within AWS. This meant that the full reads file could be stored in this system and accessed from each machine. Accordingly, the latter option to send

location pointers was chosen. This meant small modifications to the BWBBLE code which is discussed in section 5.2.

4.2. Amazon Web Services

AWS is the world's largest cloud computing platform, ahead of Google Compute Cloud, Microsoft Azure and IBM Cloud. It commands about 40% of the cloud computing market share, almost twice as much as its three biggest competitors combined (Synergy Research Group, 2017). AWS is also the platform that is most commonly used within genomic research (Kaur and Kaur, 2015). This section describes the design of the distributed cloud system that will be set up and run on AWS. The system involves a number of VMs with access to a shared file system that are controlled from a local machine.

4.2.1. Elastic Compute Cloud

EC2 is Amazon's provider of virtual machines. These machines are incredibly customizable and cost effective. They are secured using public-key cryptography meaning a key-pair is needed to access the machine. Key-pairs consist of a public key which Amazon stores and a private key that the user stores. Together, they give the user secure access to the instance. Amazon has several general-purpose instances including their t2 instances. Due to my intention to test the program using BWBBLE's built-in multithreaded option, I chose a machine with 4 vCPUs. I used a *t2.xlarge* instance which costs 17c per hour.

Name	vCPUs	RAM (GiB)	CPU Credits/hr	On-Demand Price/hr
t2.xlarge	4	16.0	54	$0.1856

Table 1 t2.xlarge instance details

It is possible to save the state of an EC2 machine for later deployment. This is known as an Amazon Machine Image. When created, an AMI will hold all installations, files and programs that are on that EC2 instance. The Amazon account will be charged for the storage space used to hold the AMI in its current state but not for the actual creation of the AMI.

4.2.2. Elastic File Storage

EFS is one of Amazon's cloud storage systems. It was designed to allow users to only pay for the storage space they are using. The storage is scalable and expands as the user adds more data to the system. EFS was chosen for this project because it can be used as a shared file system. It is possible to access a single EFS from multiple VMs concurrently making it ideal for use in a distributed system. EFS storage costs about 29c per GB per month. The test files are stored in this EFS and take up 0.54 GB, costing just 16c per month.

4.2.3. Command Line Interface

CLI is a unified tool that can be used to control and manage all of AWS. This is an alternative to using the AWS Management Console which is the browser-based console for manual control of AWS. CLI uses shell commands to control AWS from a command line or shell script. Commands exist for all possible interactions with AWS, for example, launching an EC2 instance or mounting an EFS. CLI also allows for automation of a system using shell scripts to hold a collection of commands.

4.3. Docker

Docker is an open-source computer program for OS-level virtualization (Boettiger, 2015). It is the most-popular containerization platform in the world and is one of the fastest growing new-technologies of the past few years. With this in mind, I decided to explore Docker as part of this dissertation. In particular, I wanted to see if integrating Docker into the project would have any benefits on the overall design.

4.3.1. Comparison of Docker Containers and Virtual Machines

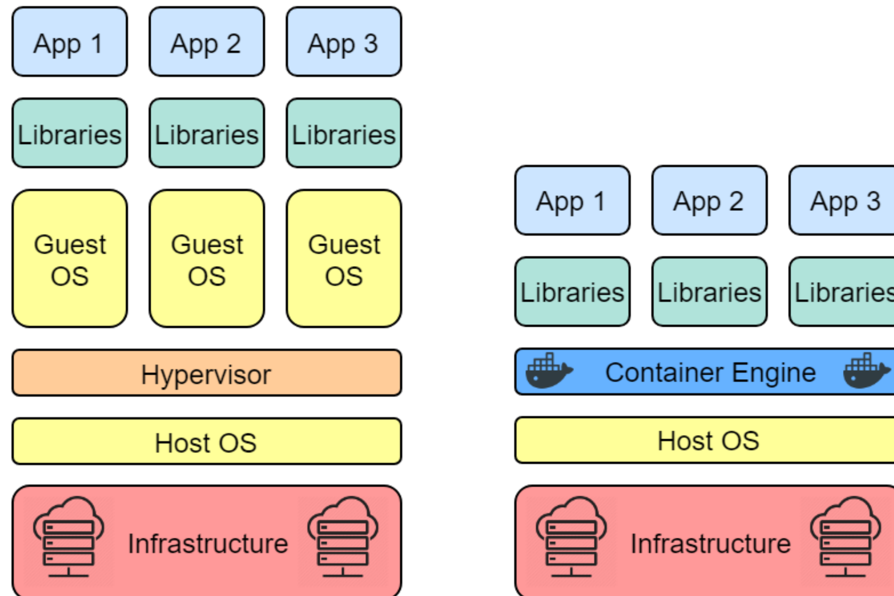

Figure 13 Comparing Virtual Machine and Docker Container Architectures

Docker is used for running applications in an isolated environment called a container. These containers are similar to virtual machines, the difference being the virtualization of the operating system. Virtual machines have their own operating system, including the kernel which is the core of every operating system. The kernel provides the file system, CPU scheduling, memory management, and other operating system functions through system calls (Silberschatz and Gagne, 2018). Comparatively, containers use the host machine's kernel and multiple containers can share the same kernel. Each container can be constrained to use a defined amount of resources available in the host machine (Preeth et al., 2015). All other aspects of containers are like that of a virtual machine. Because of this inherent difference, Docker containers start up in seconds and use less resources and memory.

4.3.2. Docker Images

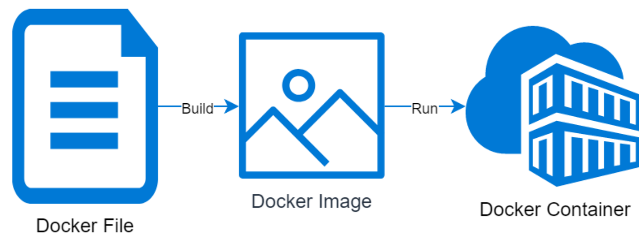

Figure 14 Docker File to Image to Container

Docker images are read-only templates which are used to run Docker containers (Preeth et al., 2015). There is an online repository (Docker Hub) where over 2 million existing images are available. About 150 of these are official Docker images and include essential OS such as Ubuntu and CentOS which are the most popular base images as they provide a good starting point for users.

Docker images can also be run by building a Dockerfile. These are plain-text files that are used to specify the steps to create the desired Docker image. They can be used to configure the OS, install necessary packages and software, set default commands to run when the container has started and more. This project makes use of an existing Ubuntu Docker image so Dockerfiles were not required.

4.3.3. Docker Bind Mount

Docker possesses a similar functionality to the mounting of an EFS on an EC2 instance. This is called a bind mount and it allows a machine to run a Docker container with a file or directory mounted onto it. This functionality is used when first running the Docker container. The mount type is specified along with the source file/directory on the machine and the target path within the container. In the case of this project, the EFS on the EC2 instance can be the source directory of the bind mount thus giving the Docker containers access to the EFS.

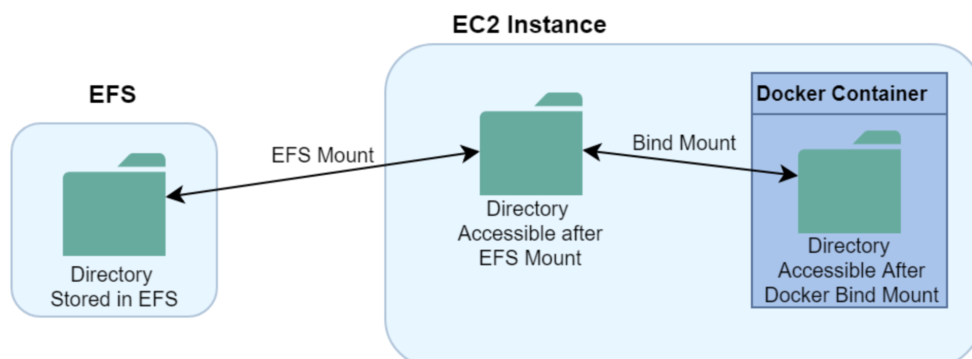

Figure 15 Showing EFS Access From Docker Container

5. Implementation

This chapter explains the implementation of the project. It describes the code modifications made to BWBBLE, the setup of the distributed system and finally, the automation script that runs AWS-BWBBLE.

5.1. Generation of Test Files

The BWBBLE GitHub repository contains a folder of files for testing the program. This includes the chromosome 21 multi-reference genome which is used during the development and testing of the AWS-BWBBLE program. It also contains a reads file with 100 reads. The problem with using this file for testing is that the file completes the read alignment process too quickly to clearly see how much of a difference is made when distributing the reads onto multiple machines. The alignment runs on every read in the file regardless of any duplicate reads. Meaning that if the same information is duplicated several times within one file, the program will take a lot longer to process the data. Using this method, I generated a file with 2,048,000 reads that took the original program almost 2 minutes to run. This made it a lot easier to evaluate the difference in run times when the distributed system is used.

5.2. Code Modification

```
Usage: bwbble align [options] <seq_fasta> <reads_fastq> <output_aln>
Options: M    mismatch penalty (default: 3)
         O    gap open penalty (default: 11)
         E    gap extend penalty (default: 4)
         n    maximum number of differences in the alignment (gaps
              and mismatches) (default: 0)
         l    length of the seed (seed := first seed_length chars
              of the read) (default: 32)
         k    maximum number of differences in the seed (default: 2)
         o    maximum number of gap opens (default: 1)
         e    maximum number of gap extends (default: 6)
         t    run multi-threaded with t threads (default: 1)
         b    indicate a start point of the reads file (must be used with -f) (default: -1)
         f    indecate an end point of the reads file (must be used with -b) (default: -1)
         S    align with a single-genome reference
         P    use pre-calculated partial alignment results
```

Figure 16 BWBBLE Updated Read Alignment Options

There is a ready-defined list of ‘options’ that can be used when running the read alignment command in BWBBLE. These options include the ability to use the multithreaded version, as discussed in section 3.2.1, as well as modifying factors such as the mismatch penalty and differences allowed in the alignment (see figure 16 for all alignment options). The original list was expanded upon to add the functionality of indicating a subset of the reads file to process. I chose the letters *‘b’* and *‘f’* to indicate *“beginning”* and *“finish”*, these were

chosen because an instruction already existed for both *'S'* and *'e'*, the desired letters for *"start"* and *"end"*.

There is an object in the program called *'aln_params_t'* which stores the alignment parameters. This object stores the default values for each of the possible options unless an option is inputted by the user. An integer was added to this type definition for both the start and end positions. When setting the default parameters, I chose to set the default of both to -1. It is good practice to set them to a negative integer since the only accepted values are positive integers. These parameters are checked when the reads file is initially processed. If both parameters contain -1, the program will run as normal. However, if they are populated with acceptable data the program will only process the reads within the specified interval. This interval includes the read at the start position and excludes the read at the end position, this is to ensure that a read will not be read twice when the code is later being run on multiple nodes. Acceptable data for these parameters are positive integers between 0 and the length of the file minus 1, with the start position being a lower value than the end position.

```
@21_32946226_32946798_3:0:0_3:0:0_0/1
ATAATCCTAGCACTTTTGGAGGCCGAGGCTGGCAGATCACTTGAGACCAGGATTTCGAGACCTGCCTGGCCAACATGGTGAAACCTCATCTCTACTAAAA
+
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
@21_16211141_16211694_4:0:0_0:0:0_1/1
TGTTTCTTTTCTTTAGGTCAGAGCCCATTAAATGATTCACTTTATGTATATCACTTAGACAATAAACATTCTATCCCTTTCCTCTCTGACTGTAAGACCA
+
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
@21_32773354_32773803_3:0:0_0:0:0_2/1
CATGGCTGCCATATCCCAATAGAGGAACAGTTTACCATTGTTTATATTCGGCTCTAGGCTACAATAGGAGGGCAGGGCCCATTTTTTCTTGCCTAGTGTT
+
```

Figure 17 Sample of reads file

Figure 17 shows a sample from the reads file used in the testing of the project. This is the standard format for these files in read alignment programs. When reading the file in BWBBLE there are defined steps for each of the four lines of each read. This meant that to get the number of reads in a file the system simply counts the number of lines in the program and divides by four. Therefore, the division of the reads using the new parameters was not too complicated and to get a read at a given point, it just must be multiplied by 4.

In the main function of the program, any parameters (options) passed with the alignment instruction are parsed with a switch statement. To allow the new options to be accepted, they had to be added to this switch statement (see appendix 1). This is where the alignment parameters are set when a user indicates an option. If the switch statement recognises one of our new options in the argument list, it converts the inputted string to an integer and stores it in the corresponding alignment parameters object.

5.3. AWS Setup

This section outlines how the components that make up the distributed network are set up. The design of the system is discussed in section 4.2 along with descriptions of each component.

5.3.1. Elastic File Storage

EFSs are created on the File Systems dashboard on the AWS Management Console. Amazon's Virtual Private Cloud is used to isolate the desired elements of the AWS cloud. The VPC for this project allows access to the *'eu-west'* availability zones. There is a VPC ID associated with every EC2 instance. The file system's VPC ID must match that of the EC2 instances that will be mounted to it. The file system's access is configured by creating mount targets in the VPCs availability zones so that EC2 instances is any of these zones will be able to mount to the EFS.

VPC	Availability Zone	Subnet	IP address	Security groups
vpc-56cbec30 (default)	eu-west-1a	subnet-25811143 (default)	Automatic	sg-18ec4967 - default
	eu-west-1b	subnet-a7fb9bef (default)	Automatic	sg-18ec4967 - default
	eu-west-1c	subnet-ec31e3b6 (default)	Automatic	sg-18ec4967 - default

Figure 18 EFS File System Access

5.3.1.1. Populating the EFS

To populate the EFS with the reads and reference genome I launched a temporary EC2 Ubuntu instance. Once running, I created a directory on the VM to hold the files. I used WinSCP on my PC to transfer the test files to the new directory on the VM. With the necessary files in a directory, I was able to mount these to the EFS using the following command:

$$sudo\ mount\ -t\ < vm_dir_path >\ -o\ tls\ < efs_id >:/ < efs_dir_path >$$

Where $-t$ stands for target and is followed by the path of the mount target and the added option $(-o)$ of TLS adds the encryption of the data when it's in transit.

5.3.2. Amazon Machine Image

It is possible to create a personalised AMI from a running EC2 instance. I wanted to have an AMI available for deployment with the modified BWBBLE program inside a Docker container. As a basis for this, I launched a *t2.xlarge* instance from the AWS Management Console using an Ubuntu 16.04.4 LTS machine image. When an instance is launched the user is prompted to create a new key pair or select an existing one. At

this point I created a new key pair and saved it on my local machine as '$aws - bwbble - key - pair$' for use with this and future instances.

5.3.2.1. Testing the Connection with EFS

On first attempt, the instance failed to mount to the EFS. This was due to the security groups for the instance and the EFS not being configured correctly. These security groups provide security at the protocol and port access level. The rules attached to the security groups allow for traffic flow which is necessary between EC2 and EFS. By default, all outbound traffic is allowed but no inbound traffic is permitted. To allow for file sharing an inbound connection is opened on the NFS port (2049). This new security group is added to both the EC2 instances and the EFS.

5.3.2.2. Docker on the EC2 Instance

To install Docker on the instance the following command was run:

$$sudo\ apt\ install\ docker.io$$

The user of the instance needs to log out of their account and then back in before Docker will work. This is to allow Docker to access the user permissions and allow the user to communicate with the Docker engine.

To create a Docker container with the AWS-BWBBLE program I initially ran an Ubuntu Docker container using the command:

$$docker\ run\ -dit\ ubuntu$$

Where $-dit$ is the concatenation of the three commands;

- $-d$: Run Docker in detached mode, meaning the container is run in the background.
- $-i$: Run Docker in interactive mode which keeps the container's STDIN open even when in detached mode.
- $-t$: Run Docker with a virtual terminal within the container.

We can view our Docker images and containers using the following commands:

Images = $docker\ images$

Containers = $docker\ ps$

5.3.2.3. AWS-BWBBLE in Docker Container

To get the AWS-BWBBLE program running inside the container, I needed to access the container's terminal. When a container is run without the $-d$ option the user automatically has access to the container's terminal. In other words, it is run in the attached mode. To change to this mode and enter a running container's terminal we use:

$$docker\ attach\ < container_name >$$

To clone the AWS-BWBBLE program inside the container I ran this set of commands:

$$apt - get\ install\ git$$

$$git\ clone\ http://gitlab.scss.tcd.ie/kmcginle/aws - bwbble.git$$

To build and run the program inside the container I had to install the 'build-essential' (GCC compilers and make utility) and 'libgomp1' (GCC OpenMP support library) packages.

With the program running successfully from the container's command line, the next goal was to get the program running inside the Docker container by sending a command from outside the container i.e. from the EC2 instance's terminal. The following Docker command exists for exactly this purpose:

$$docker\ exec\ < container_name >< command >$$

With the program successfully running inside the container I saved the state of the container as a Docker image using the command:

$$docker\ commit\ < container_name >< image_name >:< image_tag >$$

5.3.2.4. Docker Bind Mount Using EFS

The next challenge was that the EFS was mounted onto the EC2 instance, but this data was not being shared to the Docker container. As mentioned in section 4.3.3, a bind mount can be used to solve this. The following command is used to run a Docker container with a bind mount:

$$docker\ run -dit$$
$$--cpus =< num_cpus >$$
$$--name < container_name >$$
$$--mount\ type = bind, source =< src_path >, target =< target_path >$$
$$< image_name >:< image_tag >$$

5.3.2.5. Create AMI

At this point, the AWS-BWBBLE program can be run inside a container using the $docker\ exec$ command from the EC2 terminal. The reads and reference genome can be accessed in the EFS. This is the desired state of the machine. The EC2 Dashboard in the AWS Management Console displays the instances, to create an AMI of a running instance you simply select the instance and choose the option to create an image (see figure 19).

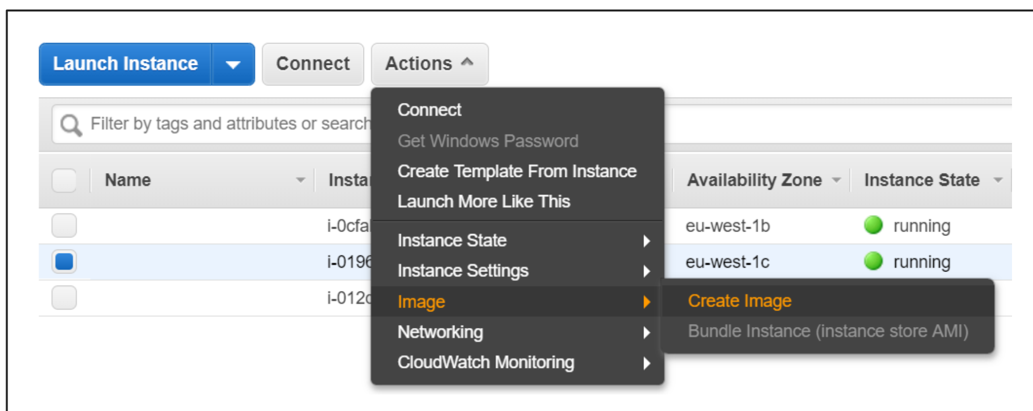

Figure 19 Create AMI in AWS Management Console

5.4. Parallelizing BWBBLE

5.4.1. Allocating CPUs in Docker and EC2

The main parallelization functionality of the AWS-BWBBLE program is achieved with the distribution of work to the EC2 instances. With this in mind, I wanted to ensure that each virtual machine was making use of the available vCPUs. As discussed, I chose *t2.xlarge* EC2 instances which have access to 4 vCPUs. The BWBBLE program is the only thing running on the VMs at any time, so it can be given full access to the vCPUs. Since the program is run within a Docker container it is also necessary to ensure that the containers have access to all available vCPUs. This is done by using the '$--cpus$' parameter when running the Docker containers.

$$\begin{aligned}&docker\ run - dit\\ &\quad --cpus =< num_cpus >\\ &\quad --name < container_name >\\ &\quad < image_name >:< image_tag >\end{aligned}$$

5.4.2. Multithreaded Version of BWBBLE

To further parallelize the system, I wanted to make use of the in-built multithreaded option in BWBBLE. As discussed in section 3.2.1, this is implemented using a read counter ($reads{->}\ count$) that is initialized when the reads file is first read and incremented for every read processed. The modification of the BWBBLE code to accept start and end positions of the reads file does not affect the functionality of the in-built multithreading. This is due to the read counter being incremented for each read processed, not each read in the file.

Appendix 2 contains the C code of the function that loads the read sequences from the file. The processed reads are stored in an array within the '$reads_t$' object. When a valid start and end position is entered, this function only processes the subset within these parameters. Since the reads counter is incremented within a conditional statement that checks that the read is inside these parameters, it will be equal to the amount of reads we intend to process whether the start and end options are being used or not.

5.5. Automation

A shell script contains commands to automate the AWS-BWBBLE program. It makes use of the AMI and EFS that were previously created. It incorporates existing Linux commands as well as many Amazon CLI commands.

5.5.1. CLI Configuration

The CLI functionality can be downloaded on any machine. I used my Windows PC to create and run the shell script for automating the project. There is a windows installer available on the Amazon website to download the CLI for use in a bash terminal on a Windows machine. When download, the CLI must be configured. This is done by running the '$aws\ configure$' command which will prompt the user for four pieces of information:

$$\boldsymbol{AWS\ Access\ Key\ ID}\ [\boldsymbol{none}]: < access_key_id >$$
$$\boldsymbol{AWS\ Secret\ Access\ Key}\ [\boldsymbol{none}]: < secret_access_key >$$
$$\boldsymbol{Default\ region\ name}\ [\boldsymbol{none}]: < region_name >$$
$$\boldsymbol{Default\ output\ format}\ [\boldsymbol{none}]: < format >$$

The access key ID and secret access key are the credentials associated with the user's AWS account. The configuration of the user's credentials is necessary before any CLI commands will be successful as each request to AWS will check that the credentials match that of the user. The default region name determines the AWS Region, all requests will be sent to this region by default. It is recommended to set this to the nearest region which in my case is $eu - west - 1$. However, it can be set to any desired region and can even be reconfigured for individual commands. The default output format determines the way any output will be presented on the terminal. The possible formats are json, text and table. The '$[none]$' indicates that this is the first time the CLI is being configured, these will get populated with the inputted values if the configure command is run again. This is a future reminder for user as to what the values are set to.

5.5.2. Shell Script

5.5.2.1. Launch Instances

The following command is used to launch EC2 instances. The $< image_id >$ is the ID of the AMI that was previously created. By using the '$count$' parameter, a number of identical machines can be created at the same time which is ideal for this system. As mentioned, the chosen instance type *t2.xlarge*. I used the '$aws - bwbble - key - pair$' that I previously downloaded to my local machine with each of the instances. Finally, the security groups must be passed to the command using their unique IDs to ensure each instance will be able to contact the EFS over the network.

$$aws\ ec2\ run - instances$$
$$- - image - id < image_id >$$
$$- - count < num_instances >$$
$$- - instance - type\ \ < instance_type >$$
$$- - key - name\ \ < private_key_name >$$
$$- - security - group - id\ \ < security_groups >$$

5.5.2.2. Get Instance Details

Some of the later commands require the instances' IPs, IDs or DNS addresses. I used the '$aws\ ec2\ describe$' command to extract this information. Below is the command being used to extract the instance IDs. The filter option accepts name-value pairs and can be used to refine the result. I used two name-value pairs to get my desired result. The first extracts the instances that match my specified instance type ('$t2.xlarge$') and the second looks at the state of these instances and returns only those that are in the process of starting up ('$pending$') or those that are already running. This allows the shell script to see all the instances we have launched even if they have not completed the start-up process when a command like this is run.

$$ids_str = \$(aws\ ec2\ describe - instances$$
$$- - filters\ "Name = instance - type, Values =< instance_type > "$$
$$- - filters\ Name = instance - state - name, Values = pending, running$$
$$- - query\ "Reservations[*].Instances[*].InstanceId"$$
$$- - output = text)$$

In the example on the previous page, 'ids_str' will equal a string of the instance IDs each separated by a tab. The same command is run with '$PrivateIpAddress$' replacing '$InstanceId$' for the IP addresses and '$PublicDnsName$' to extract the DNS addresses. The position of the data in each string corresponds to the same instance since the data is retrieved in the same manner. That is, the first IP, ID and DNS all correspond to the same instance.

5.5.2.3. Wait for Status Checks

As mentioned, some commands can run while the EC2 instances are still pending. To ensure that the instances are in the essential state before sending commands to them, the following commands are run. These ensure that the instances are in the '$running$' state and that they have passed the status checks which will identify if there are any hardware or software issues with an instance. EC2 instances will not process a command until the status checks complete so it is important to wait for these to finish.

$$aws\ ec2\ wait\ instance\ -running\ \ --instance-ids < ids_str >$$
$$aws\ ec2\ wait\ instance\ -status-ok\ \ --instance-ids < ids_str >$$

5.5.2.4. Send Command Using SSH

The following command uses an essential Linux tool which sends a command to be executed on another machine. SSH is a network protocol for operating network services securely and requires the private key, user name ('$ubuntu$') and DNS address of the machine. The command inside the quotation marks will be sent to the desired machine.

$$ssh -i < private_key_name >.pem\ ubuntu@ < dns_addr > " < cmd > "$$

A DNS address is required for each SSH command called but we often want to send the same command to all the addresses. The best way to do this is within a for loop with the DNS addresses stored in an array. Since a tab-separated string already exists with the DNS addresses, I used the internal field separator to split the string using the tab as a delimiter. The same is done to generate an array of the IP addresses.

$$IFS = \$'\backslash t'\ \ read\ -r\ -a\ < dns_array >\ <<<\ < dns_string >$$

5.5.2.5. Mount EFS

The next two commands are called from inside a for loop where 'i' is the index of the DNS array. This command ensures that each instance gets mounted onto the same EFS.

$$ssh - i < private_key_name >. pem\ ubuntu@(< dns_ar > [i]) \\ "sudo\ mount\ -t\ < vm_dir_path > -o\ tls \\ < efs_id >:/< efs_dir_path > "$$

5.5.2.6. Run Docker Container with Bind Mount

With the EFS mounted, this command starts a Docker container using the AWS-BWBBLE image and mounts the EFS to the container.

$$ssh - i < private_key_name >. pem\ ubuntu@(< dns_ar > [i]) \\ "docker\ run - dit \\ - - cpus =< num_cpus > \\ - - name < container_name > \\ - - mount\ type = bind, source =< src_path >, target =< target_path > \\ < image_name >:< image_tag > "$$

5.5.2.7. Run Reference Indexing

Some of the commands only need to be run on one of the instances. This instance will be known as the *'master'* as it provides some indirect control over the other instances. The BWBBLE index command only needs to be run on one machine because its functionality is not being parallelized and the result files will be stored in the EFS where all the other instances can access them for the read alignment.

$$ssh - i < private_key_name >. pem\ ubuntu@(< dns_ar > [0]) \\ "\ docker\ exec\ < container_name > \\ /aws - bwbble/mg - aligner/./bwbble\ index\ < efs_path >/chr21.fa "$$

5.5.2.8. Split the Reads

Before the read alignment can be run, we need to determine which portion of reads to instruct each instance to align. The first step of this is to get the total number of reads. This is done using the 'wc' Linux command which gets the word count of a file, by using the '$-l$' option we, instead, get the number of lines in the file. This number is then divided by 4 to get the number of reads since each read in the file spans 4 lines.

$$lines_in_fastq =$$
$$\$(ssh\ -i\ < private_key_name >.pem\ ubuntu@ < dns_addr >$$
$$“wc\ -l\ <\ efs/dummy_reads_large.fastq”)$$
$$num_reads = \big((lines_in_fastq/4)\big)$$

To get the number of reads for each instance, the total number of reads is divided by the number of running instances. The variables '$start$' and 'end' and created to be used within the for loop that will run the alignment command. They are both incremented by the variable '$segment_size$' after each run to point to the next section of reads in the file. Figure 20 shows a visual depiction of how these variables are used to access the correct positions in the file.

$$segment_size = \$\big((num_reads/< num_instance >)\big)$$
$$start = 0$$
$$end = \$segment_size$$

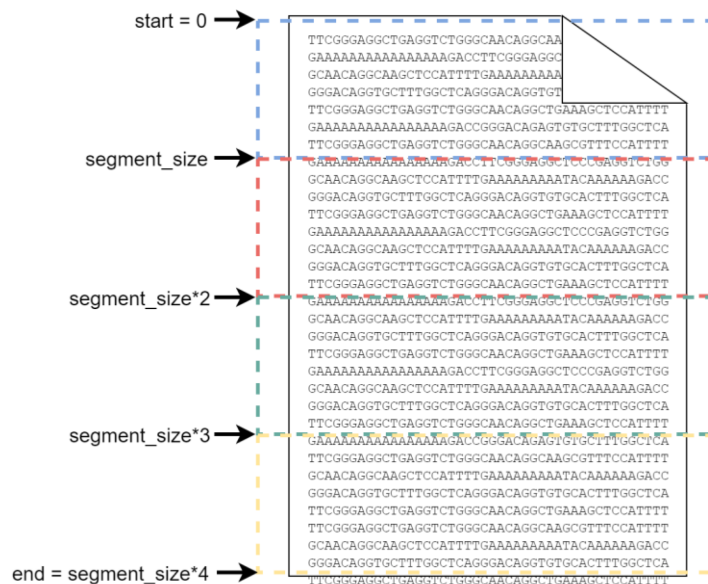

Figure 20 Depiction of the Splitting of Reads in the Shell Script

5.5.2.9. Subshells for Parallel Processing of Alignment

The following command is starting a subshell in each instance to run the alignment. It uses the $< start >$ and $< end >$ variables as described above, to evenly split the work amongst the machines. A subshell is a child process and allows the script to process the commands in parallel. To initiate a subshell, a normal SSH command just needs to be surrounded by parenthesis. The ampersand symbol is used to run the specified command in the background which means that there is no interference between the current terminal and each of the processes running on the EC2 instances. The output alignment file for each instance includes the instance IP in the file name to make it clear which instance it was generated from. A simple '$wait$' command is used to wait for all the subshells to complete.

$$
\begin{aligned}
&(echo\ \ -e\ \text{"}Subshell\ running\ alignment\ on\ \ < ip > \text{"}\ \&\\
&ssh - i\ \ < private_key_name >. pem\ ubuntu@ < dns_addr >\\
&\quad \text{"}docker\ exec\ \ < container_name >\\
&\qquad /aws - bwbble/mg - aligner/./bwbble\ align\\
&\qquad -t\ \ < num_threads >\ -b\ \ < start >\ -f\ \ < end >\\
&\qquad < efs_path >/chr21. fa\\
&\qquad < efs_path >/dummy_reads_large. fastq\\
&\qquad < efs_path >/dummy_reads_ < ip >. aln\ \text{"})\ \&
\end{aligned}
$$

5.5.2.10. Concatenate Results and Convert to SAM Format

Before converting the alignment results to a SAM file, I added a conditional check to ensure that all the alignment processes were completed successfully and that their respective alignment results were stored in the EFS. To do this I run the '$-f$' (find) command on the master node for each of the expected alignment files. This is achieved using a for loop where '$< ip >$', in turn, points to each of the IPs in the IPs array. If the file is found the variable '$exists$' is set to true and the name of the alignment file is added to an array called 'aln_files'. If, at any point, this variable is set to false it is clear that something has gone wrong during the alignment process.

$$
\begin{aligned}
&exists =\\
&\$(ssh\ \ - i\ \ < private_key_name >. pem\ ubuntu@(< dns_ar >\ \ [0])\\
&\quad \text{“[[}-f\ \ < efs_path >/\text{”}dummy_reads_ < ip >. aln\text{”]] \&\&}\\
&\quad\ echo\ true\ ||\ echo\ false;\text{”})
\end{aligned}
$$

As mentioned, when the IPs and DNSs are first retrieved they are listed in corresponding order. When they are later converted to arrays they maintain this correlation. This is very useful for the concatenation of the alignment results since everything that has been processed on all instances has been done in the order specified by the arrays. Therefore, we run the following command on the master instance using the array of file names generated in the previous step. '$aln_files[@]$' lists all elements of the array to be concatenated.

$$ssh - i < private_key_name >.pem\ ubuntu@ < dns_addr >$$
$$"sudo\ bash\ -c\ 'cat\ \ "(< aln_files > [@])"\ >$$
$$< efs_path >/concatenated.aln'"$$

The final alignment file '$concatenated.aln$' is then converted to the standard SAM format using this command:

$$ssh - i\ < private_key_name >.pem\ ubuntu@ < dns_addr >$$
$$docker\ exec\ < container_name >$$
$$/aws - bwbble/mg - aligner/./bwbble\ aln2sam$$
$$< efs_path >/chr21.fa$$
$$< efs_path >/dummy_reads_large.fastq$$
$$< efs_path >/concatenated.aln$$
$$< efs_path >/dummy_reads.sam$$

The SAM file is downloaded from the master instance to the local machine using SCP (secure copy) so that the user has access to it after the instances are shut down.

$$scp - i < private_key >.pem$$
$$ubuntu@<dns_addr>: < efs_path >/dummy_reads.sam\ .$$

5.5.2.11. Clean Up and Termination

The following command removes all the newly generated data from the EFS so that the script can be run again with the EFS in its original state with only the reads file and the reference genome.

$$ssh - i\ < private_key_name >. pem\ ubuntu@ < dns_addr >$$
$$sudo\ rm$$
$$< efs_path >/*. aln$$
$$< efs_path >/*. ann$$
$$< efs_path >/*. bwt$$
$$< efs_path >/*. ref$$
$$< efs_path >/*. sam$$

All the instances are shut down using this CLI command to ensure there isn't a large build-up of running instances.

$$aws\ ec2\ terminate - instances\ - -instance - ids\ < id_list >$$

A variation of following code was used to set up timers for each section of the code; instance setup, indexing, alignment, generation and download of SAM file and finally, the termination of the instances. The alignment timer is of great importance and is looked at closely is the results section of this dissertation.

$$timer_start = (date + \%s\%N|cut - b1 - 13)$$
$$timer_end = (date + \%s\%N|cut - b1 - 13)$$
$$timer_duration\ =\ \$((timer_end\ -\ timer_start))$$

6. Results

This chapter presents the results of the project initially using the in-built multithreaded option of BWBBLE and then discussing its drawbacks. Finally, it presents the definitive results in comparison to results from previous works.

6.1. Initial Testing

The initial testing uses the shell script outlined in section 5.5 to create the distributed system on AWS and run the AWS-BWBBLE program. It also makes use of the integrated multithreaded option of the BWBBLE program. As discussed, the chromosome 21 multi-reference genome and a FASTQ file containing 2,048,000 reads are used in all tests.

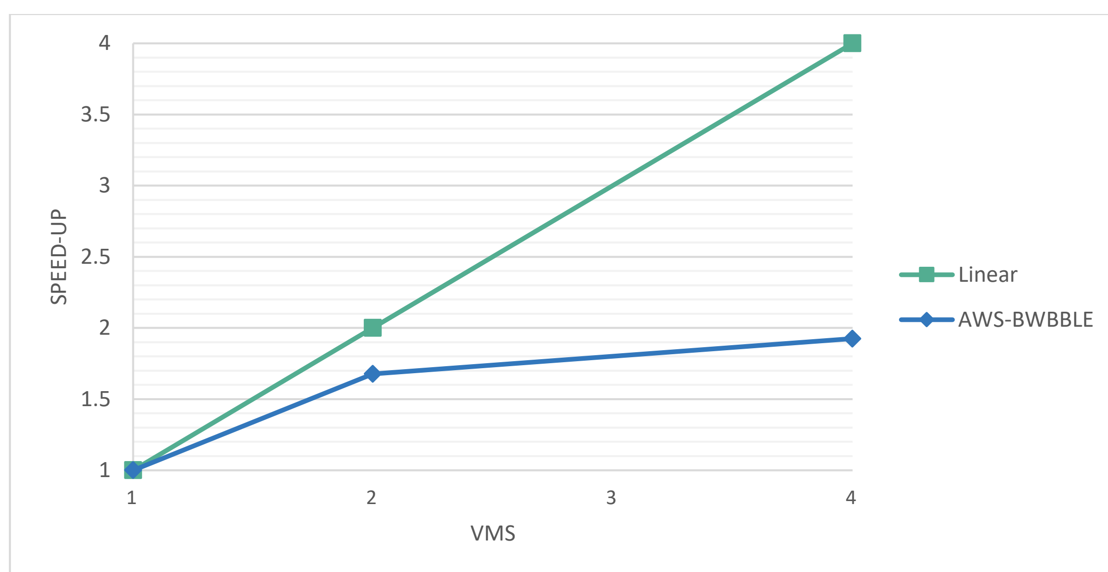

Figure 21 Speed-Up of AWS-BWBBLE Using 4 Threads

As seen in figure 21, the initial results contrasted with the expected linear speed-up. The program achieved a speed-up of 1.67 when distributing its work to 2 VMs which is not too far off the expected speed-up of 2, however, when run with 4 VMs the program only achieved a 1.9 times speed-up. At very least we should expect a speed-up of about 3.34 at this point since there should be some correlation between the speed and the number of VMs. This speed-up is less than the speed-up achieved in both SparkBWA and SparkBWBBLE which achieve 2.5 and 2.1 times speed-up with 4 times the computing power, respectively. Because of this, I suspected there may be an error somewhere in the implementation.

6.2. Testing Built-In Multithreading

It seemed that the most possible error in the program would have something to do with the in-built multithreaded option. I suspected this as there is nothing explaining the implementation or results of the option used on the original program in the journal article (Huang and Popic, 2013). The paper simply states that the feature exists but doesn't explain it any further.

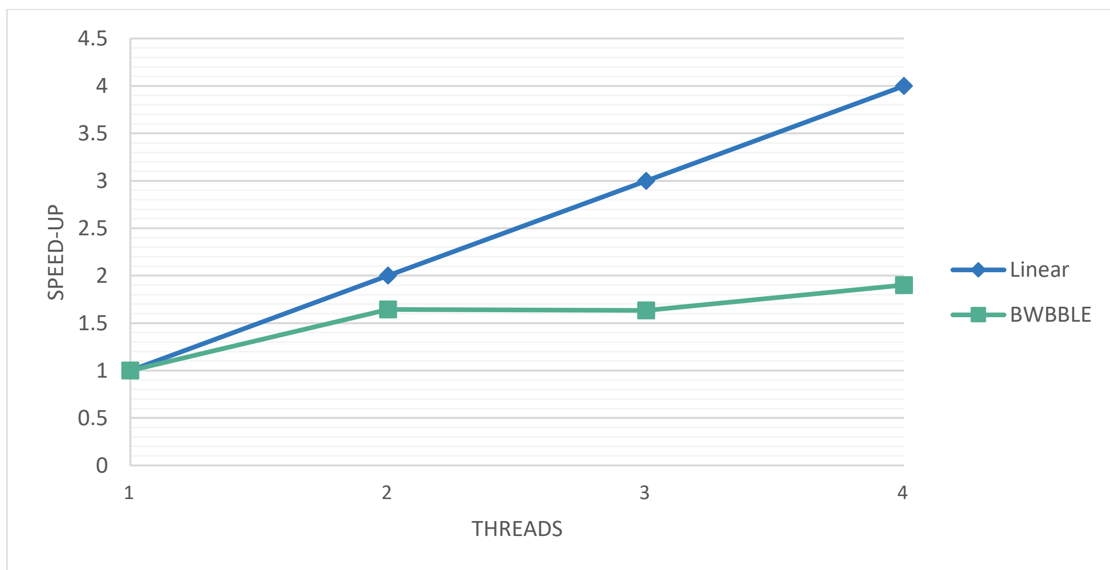

Figure 22 Comparing the Speed-Up of The Original BWBBLE Program Using the Multithreaded Option

Figure 22 shows the speed-up achieved using the multithreaded option with 1 to 4 threads on the original BWBBLE program. From the testing of this option it is clear that it is not in optimal working order. To ensure that this was the reason for the lack of speed-up in the initial tests I wanted to search for a correlation between the speed-up of BWBBLE using 'n' threads and AWS-BWBBLE using the multithreaded option and 'n' VMs.

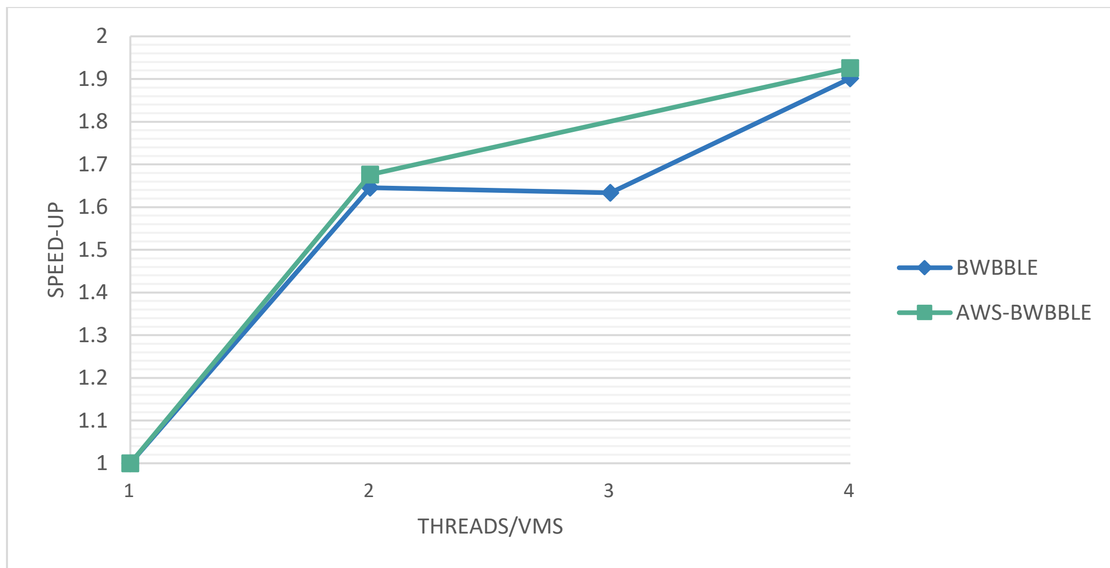

Figure 23 Linking AWS-BWBBLE's Lack of Speed-Up to BWBBLE's Threads

Figure 23 shows the speed-up of BWBBLE when run on 1, 2, 3 and 4 threads and the speed-up of AWS-BWBBLE when run on 1, 2 and 3 VMs, using 4 threads for each run. This test was done to see if there was any correlation between the previous two tests. The graph shows a strong correlation between the two tests insinuating that the lack of speed-up is due to the multithreading option being utilized. This proves that the insufficient speed-up in the initial test is indeed caused by the in-built multithreaded option.

The multithreading of the BWBBLE program is achieved using OpenMP. Upon investigation into this issue, I found one notable error in the parallel code which is that the multithreaded function is allocating and freeing the same piece of memory n times, where n is equal to the number of threads running. This would be correct if the amount of memory fit the batch size of reads processed in each thread, however, the amount of memory being allocated each time is actually the size of all the batches. The function uses the same allocation amount that is used in the non-threaded function which is the size of the entire set of reads to be processed. This appears to be the reason that the BWBBLE program has a depleting speed-up especially when considering that there is less speed-up as more threads are run and for each thread running, there is more unnecessary memory allocation. Both functions can be found in appendix 3.

6.3. Testing Different Numbers of Threads

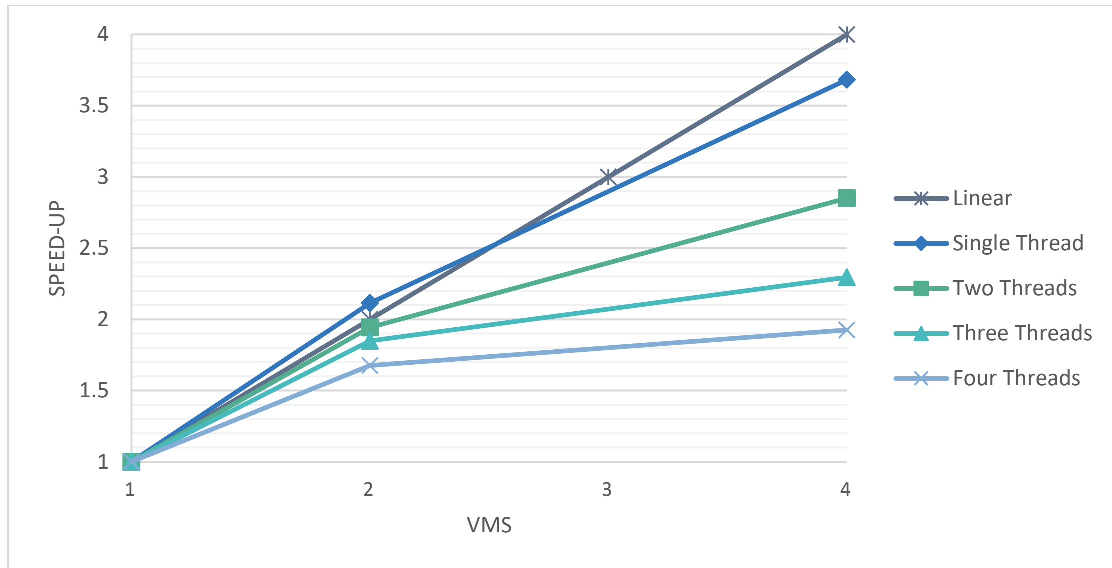

Figure 24 Comparing the Speed-Up of AWS-BWBBLE Using BWBBLE's In-Built Multithreaded Option

Figure 24 compares the speed-up times of AWS-BWBBLE using different numbers of threads. These are compared to the desired linear speed-up. The graph shows the more threads that are run corresponds to a lower speed-up which is evident from the previous tests. However, this test shows that AWS-BWBBLE run on a single thread, i.e. not using the multithreading option, achieves very close to the perfect linear speed-up.

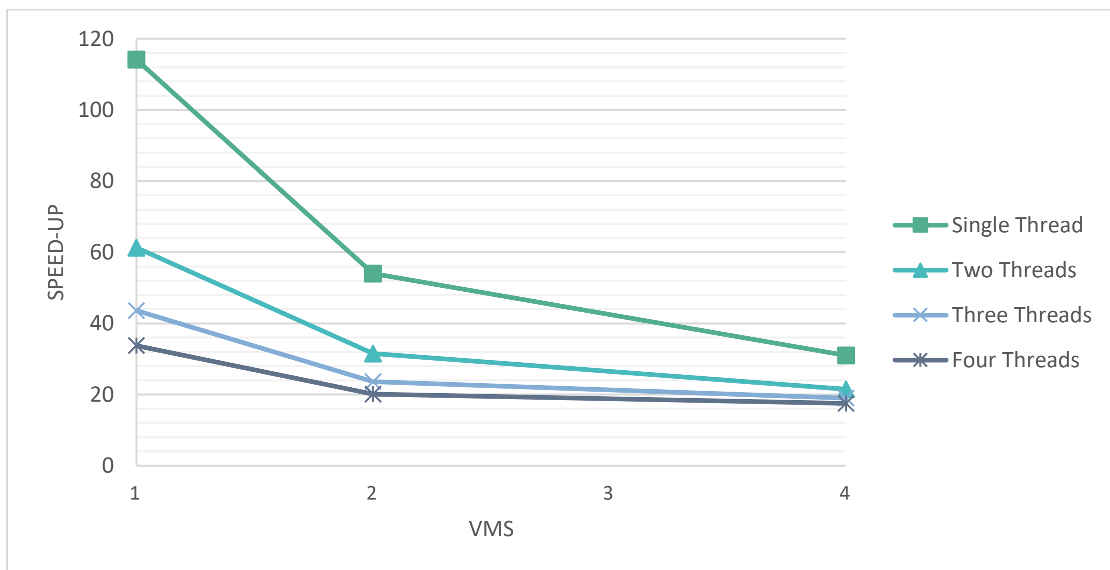

Figure 25 Run Times of Multithreaded AWS-BWBBLE

Figure 25 shows the actual run times of the previous test. This graph is to remind the reader that a better speed-up does not correlate to a better run time since all speed-up tests are based on that test's initial run time. However, each of these tests achieves a very

similar optimum run-time using 4 threads with a total of just 4 seconds separating the 2-thread and 4-thread runs.

6.4. Comparing Results to Previous Work

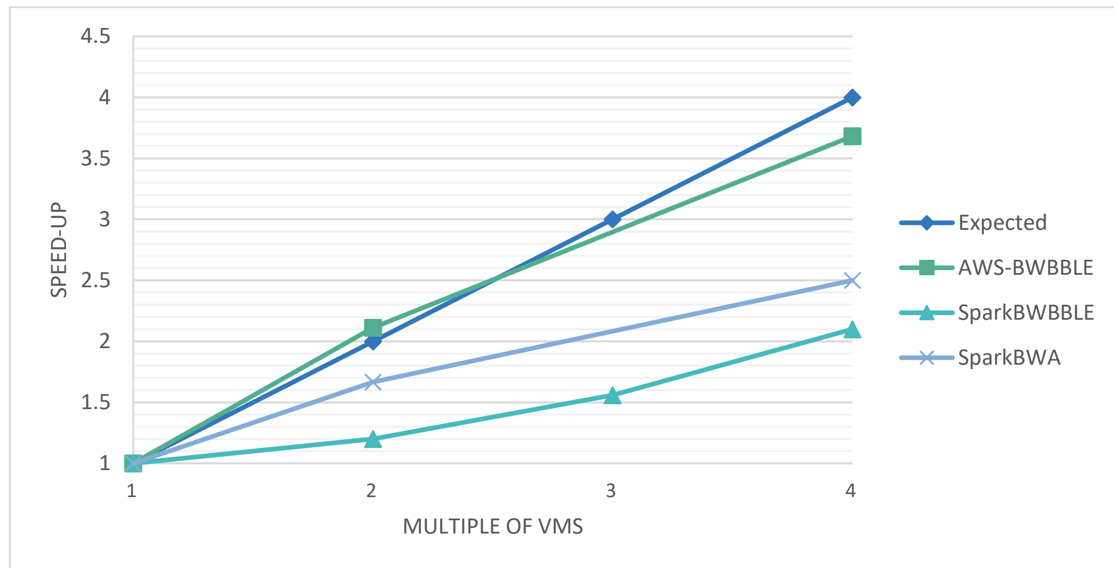

Figure 26 Comparing the Speed-Up of AWS-BWBBLE and SparkBWBBLE

Figure 26 compares the speed-up of AWS-BWBBLE (non-multithreaded) to the speed-up of SparkBWBBLE and SparkBWA. It is obvious by the graph that AWS-BWBBLE achieved much greater speed-up than the other programs. With 2 VMs, the program slightly exceeds the linear speed-up and has a greater speed-up than SparkBWBBLE and SparkBWA combined. With 4 VMs, the speed-up is slightly below linear with 3.6 times speed-up opposed to 4. This remains an impressive speed-up, especially when compared to the other programs. There is a 1.6 times speed-up between 2 VMs and 4 VMs for AWS-BWBBLE which is about twice the speed-up of the average of the other programs. At this point the project can be considered a success, however, further testing with more VMs will be necessary to ensure that the speed-up continues to grow as more and more VMs are added to the distributed system.

7. Conclusion

This chapter will conclude the paper with a discussion of the research objectives. It looks back at the research question that was posed and gives an overview of the results presented in chapter 6.

7.1. Research Question and Objectives

Chapter 1 outlines the research objectives that were undertaken to answer the posed research question which was:

"Does a straightforward approach exist that will give the BWBBLE DNA analysis program a linear speed-up when it's work is distributed amongst a number of virtual machines on the Amazon Cloud?"

The BWBBLE code was successfully modified to accept start and end positions of the reads file. This meant reading and understanding the code which was beneficial to the overall project as it meant that I had a much better understanding of how the program worked. The addition of the start and end parameters went smoothly, it involved adding code to recognize the parameters and modifying the function that reads the DNA sequences from the input file.

A shared file system was created using Elastic File Storage which was able to be mounted onto each of the VMs. The EFS was a massive benefit to the overall system as it meant that the test data did not have to be duplicated on each of the machines. The machines could read and write to the EFS directory with ease which helped the system to run smoothly. The EFS allowed the indexing and result conversions to be run on a single machine since all the alignment results were accessible on the EFS. This meant that the only thing run on the distributed VMs was the read alignment.

An Amazon Machine Image was set up using the Management Console with Ubuntu OS. Docker was incorporated in the project to investigate if it would have any benefits on the distributed network. This incorporation did not benefit the system and the inclusion of Docker complicated the EFS access and automation. That said, there were Docker features available to make everything work together efficiently, such as the bind mount and the exec command. It appears that Docker had no effect on the speed-ups of the system. Some future work could be to set up the system in the same manner but removing the Docker elements and running BWBBLE directly from the VMs.

Amazon CLI made the automation of the system a seamless process. There are commands available for everything imaginable within AWS and there is no delay between the running of a command and its execution within AWS. The CLI is arguably a better method than using the browser-based Management Console. The console can be awkward to use due to the amount of options that can be displayed on the screen at once.

The concatenation of the results file was a simple process. The tagging of the alignment files with the VMs IP addresses made it easy to ensure that all files were created. The ability to use both CLI and Linux commands within the shell script made tasks like the concatenation and download of the results file straightforward.

7.2. Overview of the Results

All the research objectives were successfully met, and the program underwent substantial testing. The results show that the straightforward approach was successful with regards to the distribution of work on the system. The project encountered an issue with the built-in multithreaded option in BWBBLE. Although disappointing, this was an interesting discovery and could explain why the BWBBLE paper does not go into any detail about the design or usage of the option.

AWS-BWBBLE was evaluated using the multithreaded and non-threaded options. The program reached a linear speed-up when using the non-threaded version with the work distributed on up to 4 VMs. This is the best speed-up that has been achieved compared to any of the other parallelization approaches discussed in this dissertation. The testing could, unfortunately, not be taken any further due to the restrictions to the AWS account. Future work could be done to bring the testing further to see if the speed-up continues when the program is run on a much larger number of VMs.

7.3. Final Remarks

The area of bioinformatics was found to be incredibly interesting. The use of computer science in areas such as genetics can have a real effect on the world. DNA sequencing and analysis are very complex procedures but the inadequate speeds of read alignment are largely due to having too much data to process. This embarrassingly parallel problem has now been proven to be optimizable using computer systems and shows that a linear speed-up is possible for alignment programs.

8. Bibliography

ABUÍN, J. M., PICHEL, J. C., PENA, T. F. & AMIGO, J. 2015. BigBWA: Approaching The Burrows–Wheeler Aligner to Big Data Technologies. *Bioinformatics,* 31.

ABUÍN, J. M., PICHEL, J. C., PENA, T. F. & AMIGO, J. 2016. SparkBWA: Speeding Up the Alignment of High-Throughput DNA Sequencing Data. *PloS one*.

ARRAM, J., KAPLAN, T., LUK, W. & JIANG, P. 2017. Leveraging FPGAs for Accelerating Short Read Alignment. *IEEE/ACM Trans. Comput. Biol. Bioinformatics,* 14.

BOETTIGER, C. 2015. An Introduction to Docker for Reproducible Research.

BUYYA, R., BROBERG, J. & GOSCINSKI, A. 2011. Introduction to Cloud Computing. *Cloud Computing: Principals and Paradigms.*

DALE, J. W., VON SCHANTZ, M. & PLANT, N. 2012. *From Genes to Genomes: Concepts and Applications of DNA Technology,* University of Surrey, UK, Wiley-Blackwell.

FILION, G. 2016. A Tutorial on Burrows-Wheeler Indexing Methods: Part 1. 18/10/2018].

FILION, G. 2017. A Tutorial on Burrows-Wheeler Indexing Methods: Part 2. 26/10/2018].

HUANG, L. & POPIC, V. 2013. Short read alignment with populations of genomes. *Bioinformatics (Oxford, England),* 29.

HUANG, L. & POPIC, V. 2015. BWBBLE. https://github.com/viq854/bwbble.

HUMAN GENOME SEQUENCING CONSORTIUM, I. 2004. Finishing the euchromatic sequence of the human genome.

KAUR, S. & KAUR, S. 2015. Genomics with Cloud Computing. *International Journal of Scientific & Technology Research,* 4.

LANGMEAD, B., TRAPNELL, C., POP, M. & SALZBERG, S. L. 2009. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome biology,* 10.

LEE, T. F. 1991. *The Human Genome Project: Cracking the Genetic Code of Life*, Plenum Press.

LI, H. & DURBIN, R. 2009. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics (Oxford, England),* 25.

LI, R., LI, Y., KRISTIANSEN, K. & WANG, J. 2008. SOAP: Short Oligonucleotide Alignment Program. *Bioinformatics,* 24.

LI, R., YU, C., LI, Y., LAM, T.-W., YIU, S.-M., KRISTIANSEN, K. & WANG, J. 2009. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics (Oxford, England),* 25.

MAHAJAN, S. & SHAH, S. 2013. Distributed Computing. 2.

PREETH, E. N., MULERICKAL, F. J. P., BIJU, P. & YEDHU, S. 2015. *Evaluation of Docker Containers Based on Hardware Utilization*.

REHMAN, T. B. 2018. *Cloud Computing Basics*, Bloomfield: Mercury Learning & Information.

RUSSEL, P. J. 2010. *iGenetics: A Molecular Approach,* San Francisco, Calif. ; London, Pearson Benjamin Cummings.

SILBERSCHATZ, A. & GAGNE, G. 2018. *Operating System Concepts*.

STRATFORD, B. 2018. *Cloud-based high speed parallel DNA read alignment using the Burrows-Wheeler Transform.* B.A Computer Science, Trinity College Dublin.

SYNERGY RESEARCH GROUP 2017. Microsoft, Google and IBM Public Cloud Serge is at Expense of Smaller Providers.

THE GENOMES PROJECT, C. 2015. A Global Reference For Human Genetic Variation. *Nature,* 526**,** 68.

9. Appendices

9.1. Appendix 1: Setting BWBBLE Options Based on Alignment Parameters

```
while ((c = getopt(argc-1, argv+1, "M:O:E:n:k:o:e:l:m:t:b:f:SP")) >= 0) {
        switch (c) {
                case 'M': params->mm_score = atoi(optarg); break;
                case 'O': params->gapo_score = atoi(optarg); break;
                case 'E': params->gape_score = atoi(optarg); break;
                case 'n': params->max_diff = atoi(optarg); break;
                case 'k': params->max_diff_seed = atoi(optarg); break;
                case 'o': params->max_gapo = atoi(optarg); break;
                case 'e': params->max_gape = atoi(optarg); break;
                case 'l': params->seed_length = atoi(optarg); break;
                case 'm': params->max_entries = atoi(optarg); break;
                case 't': params->n_threads = atoi(optarg); break;
                case 'S': params->is_multiref = 0; break;
                case 'P': params->use_precalc = 1; break;
                case 'b': params->reads_begin_pos = atoi(optarg); break;
                case 'f': params->reads_end_pos = atoi(optarg); break;
                case '?': align_usage(); return 1;
                default: return 1;
        }
}
```

9.2. Appendix 2: Function to Load The Read Sequences from The FASTQ File

```
reads_t* fastq2reads(const char *readsFname, aln_params_t* params) {
	//printf("func: fastq2reads @ io.c\n");
	FILE *readsFile = (FILE*) fopen(readsFname, "r");
	if (readsFile == NULL) {
		printf("load_reads_fastq: Cannot open reads file: %s !\n", readsFname);
		exit(1);
	}
	reads_t *reads = (reads_t*) calloc(1, sizeof(reads_t));
	int allocatedReads = NUM_READS_ALLOC;
	reads->reads = (read_t*) malloc(allocatedReads*sizeof(read_t));
	reads->count = 0;

	// check that if begin and end pos have been set by user
	// then begin < end && begin and end are positive && within the size of the file
	if((params->reads_begin_pos != -1 && params->reads_end_pos != -1) &&
		( (params->reads_begin_pos > params->reads_end_pos) ||
			(params->reads_begin_pos < 0 && params->reads_end_pos < 0)  )) {
		// ERROR
		fprintf(stderr, "fastq2reads: Invalid start and end positions: Start = %d, "
			+ " End = %d\n",
		params->reads_begin_pos, params->reads_end_pos);
		exit(1);
	}

	int read_num = 0;
	char c;
	while(!feof(readsFile)) {
		if( (params->reads_begin_pos == -1 && params->reads_end_pos == -1) ||
		(read_num >= params->reads_begin_pos*4 && read_num < params->reads_end_pos*4) ) {

			if (reads->count >= allocatedReads) {
				allocatedReads += NUM_READS_ALLOC;
				reads->reads = (read_t*) realloc(reads->reads,
					allocatedReads*sizeof(read_t));
			}

			read_t* read = &(reads->reads[reads->count]);

			c = (char) getc(readsFile);
			while(c != '@' && !feof(readsFile)) {
				c = (char) getc(readsFile);
			}
			if(feof(readsFile)) break;

			// line 1 (@ ...)
			int seqNameLen = 0;
			c = (char) getc(readsFile);
			while(c != '\n' && seqNameLen < MAX_SEQ_NAME_LEN && !feof(readsFile)){
				read->name[seqNameLen] = c;
				seqNameLen++;
				c = (char) getc(readsFile);
			}
			read->name[seqNameLen] = '\0';
			if(feof(readsFile)) fastq_error(readsFname);

			while (c != '\n' && !feof(readsFile)) {
				c = (char) getc(readsFile);
			}
			if(feof(readsFile)) fastq_error(readsFname);

			// line 2 (sequence letters)
			int readLen = 0;
			int allocatedReadLen = READ_LENGTH_ALLOC;
			read->seq = (char*) malloc(allocatedReadLen*sizeof(char));
			read->rc = (char*) malloc(allocatedReadLen*sizeof(char));
			read->qual = (char*) malloc(allocatedReadLen*sizeof(char));
```

```
				c = (char) getc(readsFile);
				while (c != '\n' && !feof(readsFile)) {
					if (readLen >= allocatedReadLen) {
						allocatedReadLen += READ_LENGTH_ALLOC;
						read->seq = (char*) realloc(read->seq,
							allocatedReadLen*sizeof(char));
						read->rc = (char*) realloc(read->rc,
							allocatedReadLen*sizeof(char));
						read->qual = (char*) realloc(read->qual,
							allocatedReadLen*sizeof(char));
					}
					read->seq[readLen] = nt4_table[(int) c];
					c = (char) getc(readsFile);
					readLen++;
				}
				read->len = readLen;
				if(feof(readsFile)) fastq_error(readsFname);

				while (c != '+' && !feof(readsFile)) {
					c = (char) getc(readsFile);
				}
				if(feof(readsFile)) fastq_error(readsFname);

				// line 3 (+ ...)
				while(c != '\n' && !feof(readsFile)){
					c = (char) getc(readsFile);
				}
				if(feof(readsFile)) fastq_error(readsFname);

				// line 4 (quality values)
				int qualLen = 0;
				c = (char) getc(readsFile);
				while(c != '\n' && !feof(readsFile)) {
					if(qualLen <= readLen) {
						read->qual[qualLen] = c;
					}
					qualLen++;
					c = (char) getc(readsFile);
				}
				if(qualLen != readLen) {
					printf("Error: The number of quality score symbols does not match"
						+ " the length of the read sequence.\n");
					exit(1);
				}
				read->qual[qualLen] = '\0';

				// compute the reverse complement
				for(int i = 0; i < read->len; i++) {
					read->rc[read->len-1-i] = nt4_complement[(int)read->seq[i]];
				}

				if(read->len > reads->max_len) {
					reads->max_len = read->len;
				}
				reads->count++;

			} else if (read_num >= params->reads_end_pos*4) {
				break;
			}
			read_num+=4;
		}
		printf("Loaded %d reads from %s.\n", reads->count, readsFname);

		fclose(readsFile);
		return reads;
	}
```

9.1. Appendix 3: Sequential and Parallel Inexact Matching Functions

```
int align_reads_inexact(bwt_t *BWT, reads_t* reads, sa_intv_list_t* precalc_sa_intervals_table,
aln_params_t* params, char* alnFname) {
        //printf("func: align_reads_inexact @ inexact_match.c\n");
        printf("BWBBLE Inexact Alignment...\n");
        FILE* alnFile = (FILE*) fopen(alnFname, "a+b");
        if (alnFile == NULL) {
                printf("align_reads_inexact: Cannot open ALN file: %s!\n", alnFname);
                perror(alnFname);
                exit(1);
        }
        // lower bound on the number of differences at each position in the read
        diff_lower_bound_t* D = (diff_lower_bound_t*) calloc(reads->max_len+1,
                sizeof(diff_lower_bound_t));
        // lower bound for the read seed positions
        diff_lower_bound_t* D_seed = (diff_lower_bound_t*) calloc(params->seed_length+1,
                sizeof(diff_lower_bound_t));
        priority_heap_t* heap = heap_init(params);
        // process the reads in batches
        int num_processed = 0;
        while(num_processed < reads->count) {
                clock_t t = clock();
                int batch_size = ((reads->count - num_processed) > READ_BATCH_SIZE ) ? READ_BATCH_SIZE
                        : (reads->count - num_processed);
                for(int i = num_processed; i < num_processed + batch_size; i++) {
                        read_t* read = &reads->reads[i];
                        read->alns = init_alignments();
                        sa_intv_list_t* precalc_sa_intervals = NULL;
                        if(params->use_precalc) {
                                // discard reads that have N's in the last PRECALC_INTERVAL_LENGTH
                                // bases (<0 result from read_index)
                                int read_index = read2index(read->rc, read->len);
                                if(read_index < 0) {
                                        continue;
                                }
                                precalc_sa_intervals = &(precalc_sa_intervals_table[read_index]);
                        }
                        // align read with forward reference <=> read reverse
                        // complement with BWT reverse complement
                        // align read with reverse complement reference <=> read reverse
                        // complement with BWT forward
                        calculate_d(BWT, read->seq, read->len, D, params);
                        if(params->seed_length && read->len > params->seed_length) {
                                calculate_d(BWT, read->seq, params->seed_length, D_seed, params);
                        }
                        inexact_match(BWT, read->rc, read->len, heap, precalc_sa_intervals, params, D,
                                D_seed, read->alns);
                }
                printf("Processed %d reads. Inexact matching time: %.2f sec.",
                        num_processed+batch_size, (float)(clock() - t) / CLOCKS_PER_SEC);
                // write the results to file
                clock_t ts = clock();
                for(int i = num_processed; i < num_processed + batch_size; i++) {
                        read_t* read = &reads->reads[i];
                        alns2alnf_bin(read->alns, alnFile);
                        free_alignments(read->alns);
                        free(read->seq);
                        free(read->rc);
                        free(read->qual);
                        read->seq = read->rc = read->qual = NULL;
                }
                printf("Storing results time: %.2f sec\n", (float)(clock() - ts) / CLOCKS_PER_SEC);
                num_processed += batch_size;
        }

        free(D);
        free(D_seed);
        heap_free(heap);
        fclose(alnFile);
        return 0;
}
```

```
int align_reads_inexact_parallel(bwt_t *BWT, reads_t* reads, sa_intv_list_t*
precalc_sa_intervals_table, aln_params_t* params, char* alnFname) {
	//printf("func: align_reads_inexact_parallel @ inexact_match.c -- naive
	//parallelization scheme (1 thread <=> 1 read)\n");
	printf("BWT-SNP Inexact Alignment...\n");
	FILE* alnFile = (FILE*) fopen(alnFname, "a+");
	if (alnFile == NULL) {
		printf("align_reads_inexact: Cannot open ALN file: %s!\n", alnFname);
		perror(alnFname);
		exit(1);
	}
	// process the reads in batches
	int num_processed = 0;
	while(num_processed < reads->count) {
		clock_t t = clock();
		int batch_size = ((reads->count - num_processed) > READ_BATCH_SIZE ) ? READ_BATCH_SIZE
		: (reads->count - num_processed);

		omp_set_num_threads(params->n_threads);
		int tid, n_threads, chunk_start, chunk_end;
		diff_lower_bound_t* D, * D_seed;
		priority_heap_t* heap;
		#pragma omp parallel private(tid, n_threads, chunk_start, chunk_end, D, D_seed, heap)
		{
			tid = omp_get_thread_num();
			n_threads = omp_get_num_threads();
			chunk_start = tid * batch_size / n_threads;
			chunk_end = (tid + 1) * batch_size / n_threads;

			// lower bound on the number of differences at each position in the read
			D = (diff_lower_bound_t*) calloc(reads->max_len+1,
				sizeof(diff_lower_bound_t));
			// lower bound for the read seed positions
			D_seed = (diff_lower_bound_t*) calloc(params->seed_length+1,
				sizeof(diff_lower_bound_t));
			// partial alignments min-heap
			heap = heap_init(params);

			for (int i = num_processed + chunk_start; i < num_processed + chunk_end; i++){
				read_t* read = &reads->reads[i];
				read->alns = init_alignments();
				sa_intv_list_t* precalc_sa_intervals = NULL;
				if(params->use_precalc) {
				    // discard reads that have N's in the last
				    // PRECALC_INTERVAL_LENGTH bases (<0 result from read_index)
				    int read_index = read2index(read->rc, read->len);
				    if(read_index < 0) {
				        continue;
				}
				    precalc_sa_intervals = &(precalc_sa_intervals_table[read_index]);
				}
				// align read with forward reference <=> read reverse
				// complement with BWT reverse complement
				// align read with reverse complement reference <=> read reverse
				// complement with BWT forward
				calculate_d(BWT, read->seq, read->len, D, params);
				if(params->seed_length && read->len > params->seed_length) {
				    calculate_d(BWT, read->seq, params->seed_length, D_seed, params);
				}
				inexact_match(BWT, read->rc, read->len, heap, precalc_sa_intervals,
					params, D, D_seed, read->alns);
			}
			free(D);
			free(D_seed);
			heap_free(heap);
		}
```

```
printf("Processed %d reads. Inexact matching time: %.2f sec.",
                        num_processed+batch_size, (float)(clock() - t) / CLOCKS_PER_SEC);
                // write the results to file
                clock_t ts = clock();
                for(int i = num_processed; i < num_processed + batch_size; i++) {
                        read_t* read = &reads->reads[i];
                        alns2alnf_bin(read->alns, alnFile);
                        free_alignments(read->alns);
                        free(read->seq);
                        free(read->rc);
                        free(read->qual);
                        read->seq = read->rc = read->qual = NULL;
                }
                printf("Storing results time: %.2f sec\n", (float)(clock() - ts) / CLOCKS_PER_SEC);
                num_processed += batch_size;
        }
        fclose(alnFile);
        return 0;
}
```

9.2. Appendix 4: Results Data Tables

Number of Threads	**Number of Virtual Machines**		
	1	**2**	**4**
1	114.2	54.0	31
2	61.3	31.5	21.5
3	43.6	23.6	19
4	33.7	20.1	17.5

Table 2 Run Times of AWS-BWBBLE

Number of Threads	**Number of Virtual Machines**		
	1	2	4
1	1	2.1	3.7
2	1	1.9	2.9
3	1	1.8	2.3
4	1	1.7	1.9

Table 3 Speed-Up of AWS-BWBBLE

Threads	**1**	**2**	**3**	**4**
Speed-Up	1	1.65	1.63	1.9

Table 4 Speed-Up of Multithreaded BWBBLE