1-31-2018 11:00 AM

Efficient Alignment Algorithms for DNA Sequencing Data

Nilesh Vinod Khiste
The University of Western Ontario

Supervisor
Lucian Ilie
The University of Western Ontario
NA
The University of Western Ontario
NA
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

Abstract

The DNA Next Generation Sequencing (NGS) technologies produce data at a low cost, enabling their application to many ambitious fields such as cancer research, disease control, personalized medicine etc. However, even after a decade of research, the modern aligners and assemblers are far from providing efficient and error free genome alignments and assemblies respectively. This is due to the inherent nature of the genome alignment and assembly problem, which involves many complexities. Many algorithms to address this problem have been proposed over the years, but there still is a huge scope for improvement in this research space.

Many new genome alignment algorithms are proposed over time and one of the key differentiators among these algorithms is the efficiency of the genome alignment process. I present a new algorithm for efficiently finding Maximal Exact Matches ($MEMs$) between two genomes: E-MEM (Efficient computation of maximal exact matches for very large genomes). Computing MEMs is one of the most time consuming step during the alignment process. E-MEM can be used to find MEMs which are used as seeds in a genome aligner to increase its efficiency. The E-MEM program is the most efficient algorithm as of today for computing MEMs, and it surpasses all competitors by large margins.

There are many genome assembly algorithms available for use, but none produces perfect genome assemblies. It is important that assemblies produced by these algorithms are evaluated accurately and efficiently.This is necessary to make the right choice of the genome assembler to be used for all the downstream research and analysis. A fast genome assembly evaluator is a key factor when a new genome assembler is developed, to quickly evaluate the outcome of the algorithm. I present a fast and efficient genome assembly evaluator called LASER (Large genome ASsembly EvaluatoR), which is based on a leading genome assembly evaluator QUAST, but significantly more efficient both in terms of memory and run time.

The NGS technologies limit the potential of genome assembly algorithms because of short read lengths and nonuniform coverage. Recently, third generation sequencing technologies have been proposed which promise very long reads and a uniform coverage. However, this

technology comes with its own drawback of high error rate of 10 - 15% consisting mostly of indels. The long read sequencing data are useful only after error correction obtained using self read alignment (or read overlapping) techniques. I propose a new self read alignment algorithm for Pacific Biosciences sequencing data: HISEA (Hierarchical SEed Aligner), which has very high sensitivity and precision as compared to other state-of-the-art aligners. HISEA is also integrated into Canu assembly pipeline. Canu+HISEA produces better assemblies than Canu with its default aligner MHAP, at a much lower coverage.

Keywords: Bioinformatics, De novo genome assembly, E-MEM, Genome assembly evaluation, HISEA, HiSeq, Illumina, LASER, Long read alignment, Maximal exact matches, Next-generation sequencing, Pacific Biosciences, Sequence alignment

It has been a very good experience coming back to academia after a long stint in industry. However, it was a very tough decision to leave behind a set career path. I could not have done it without the support of my family. I dedicate this work to them.

Acknowlegements

I would like to express my sincere gratitude to everyone involved in the completion of this dissertation. Most importantly, I would like to thank my supervisor Dr. Lucian Ilie for introducing me to the amazing research field of Bioinformatics and string algorithms. His continuous encouragement helped me to explore various interesting problems in Bioinformatics and use my research to propose efficient solutions for some of them. I really appreciate his extraordinary patience in reviewing my manuscripts, research proposals, presentations and this dissertation. His critical feedback and suggestions were always helpful in enhancing the quality of these documents. I am very grateful to my supervisor for the valuable time he spent discussing my ideas and guiding me.

I am thankful to the Department of Computer Science at the University of Western Ontario for awarding me with the Western Graduate Research Scholarship (WGRS). I am also grateful to the Ministry of Training, Colleges and Universities (Ontario) for awarding me the Ontario Graduate Scholarship (OGS) for two consecutive years.

I am also thankful to my examiners and members of my supervisory committee Dr. Ming Li, Dr. Kathleen Hill, Dr. Roberto Solis-Oba, Dr. Anwar Haque and Dr. Kaizhong Zhang. I am thankful to my colleagues Dr. Mike Molnar and Yiwei Li for all the technical discussions around various topics. I would like to thank my friends and family members for their constant support and encouragement which kept me going and led to the completion of this research work, which is one of the biggest accomplishments of my life.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

It has been known for decades that evolution is a change in inherited characteristics of biological populations. In 1859, Charles Darwin was the first person to propose a scientific theory of evolution which was associated with the population genetics theory. Population genetics is the study of the frequency and interactions of *alleles* and *genes* in populations. A gene is the molecular unit of heredity of a living organism. The genetic information within a particular gene may not be the same for different organisms and therefore different copies of a gene may give different instructions. Each unique form of a single gene is called an *allele*. As an example, one allele for the gene for hair color could instruct the body to produce a lot of pigment, producing black hair, while a different allele of the same gene might give garbled instructions that fail to produce any pigment, giving white hair.

Genes are made from a long molecule called DNA (Deoxyribonucleic acid), which is copied and inherited across generations. DNA is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses. DNA molecules consist of a double stranded structure forming a double helix. The two DNA strands are known as polynucleotides since they are composed of simpler units called nucleotides. Each nucleotide is composed of a nitrogen-containing nucleobase, either guanine (G), adenine (A), thymine (T), or cytosine (C) as well as a monosaccharide sugar called de-

oxyribose and a phosphate group. The nucleotides are joined to one another in a chain by covalent bonds between the sugar of one nucleotide and the phosphate of the next, resulting in an alternating sugar-phosphate backbone. According to base pairing rules (A with T and C with G), hydrogen bonds bind the nitrogenous bases of the two separate polynucleotide strands to make double-stranded DNA. The two strands of DNA run in opposite directions. The two ends of a single stranded DNA in forward direction are identified by 5' (five prime) and 3' (three prime) ends. Figure 1.1 shows a DNA molecule structure with A-T and G-C hydrogen bonds.

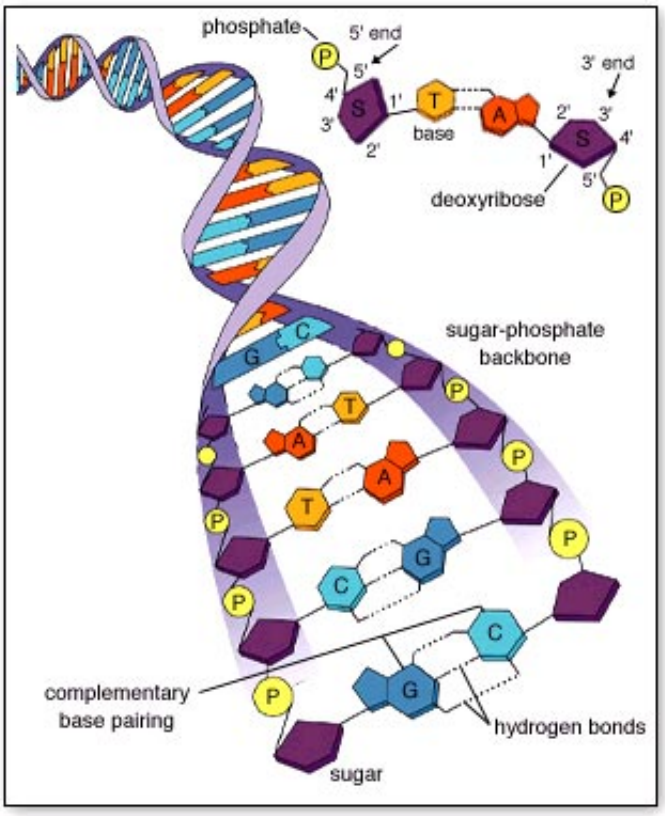

Figure 1.1: DNA molecule structure [7]

DNA sequencing is the process of determining the precise order of nucleotides within a DNA molecule. It includes any method that is used to determine the order of the four bases in a strand of DNA. The advent of rapid DNA sequencing methods has greatly accelerated biological and medical research and discovery.

1.1 DNA sequencing

The attempt for gene sequencing started in the early 1970s and the first known successful gene sequencing was done by Gilbert and Maxam [23] in 1973. They were able to produce a gene sequence of 24 base pairs by a method known as wandering-spot analysis. The method was very labor intensive and time consuming when used with real biological datasets, given the extreme length of biological sequences. This situation changed when Frederick Sanger developed several faster, more efficient techniques to sequence DNA. Frederick Sanger's work [51] in this area was groundbreaking and it led to his being a recipient of the Nobel Prize in Chemistry in 1980. Although the basic Sanger method was still being used to sequence whole genomes, time and cost considerations continued to make it expensive, thus limiting the number of genomes that could be completed. Using Sanger sequencing, the Human Genome Project took more than 10 years and incurred a cost of nearly $3 billion [1].

1.1.1 Sanger method

The Sanger sequencing method [51], also know as chain-termination method requires a single-stranded DNA template, a DNA primer, a DNA polymerase, normal deoxynucleotidetriphosphates (dNTPs), and modified di-deoxynucleotidetriphosphates (ddNTPs) for sequencing to be performed. In this method, the templates are divided into four separate sequencing reactions containing DNA polymerase and all four dNTPs. Only one ddNTP is added to each sample which produces many DNA polymerized sequences of varying length each ending in corresponding ddNTP. These polymerized sequences can then be separated by a technique called *gel-electrophoresis* based on sequence length. The base identification is performed by denaturing polyacrylamide-urea gel with each of the four reactions. The terminal ddNTP indicates whether an A, T, G, or C occurs in that position on the template strand. When a gel is stained with a DNA-binding dye, the DNA fragments can be seen as bands, each representing a group of same-sized DNA fragments. The DNA bands are then visualized by radiography or UV

light.

Over the years, this process has been greatly simplified. The sequencing can be performed in a single reaction. The chain termination based kits are commercially available which are ready to use and contain all the reagents needed for sequencing. The DNA polymerized sequences are terminated by incorporating fluorescently labeled nucleotide thereby producing a DNA ladder which makes it easier to determine the DNA sequence of the original DNA fragment. The process is repeated many times until it is guaranteed that a ddNTP is incorporated at every single position of template DNA. At this point, the samples contain fragments of different lengths, ending at each of the nucleotide positions in the DNA template. Figure 1.2 shows a comparison between two sequencing techniques - a gel based radioactive method and a fluorescent method. The image on the left is an example of the Sanger method using the gel-electrophoresis ladder. The image on the right is the automated Sanger method using florescent labels through a capillary tube [53]. The short fragments move faster and pass through the capillary tube first followed by longer fragments. The smallest fragment is created by incorporating only one nucleotide after the primer. The color of the dye is used for detection of nucleotide at the end of the tube, which produces original DNA sequence - one nucleotide at a time.

The Sanger method was expensive and time consuming due to usage of gels. It limited the application of this method to small viruses and bacterial genomes. New sequencing methods [54] utilizing automated reloading of the capillaries with polymer matrix instead of slab gels were introduced to speed up the sequencing process. The automated sequencing methods increased total throughput and decreased costs gradually for Sanger sequencing. These automated methods were successfully employed in the human genome project and reduced both cost and time required to complete the project.

Sanger sequencing provided remarkable opportunities to life sciences and improved the knowledge and understanding of cellular mechanisms and diseases. However, limitations including throughput, speed and sequencing quality remained big obstacles in its widespread

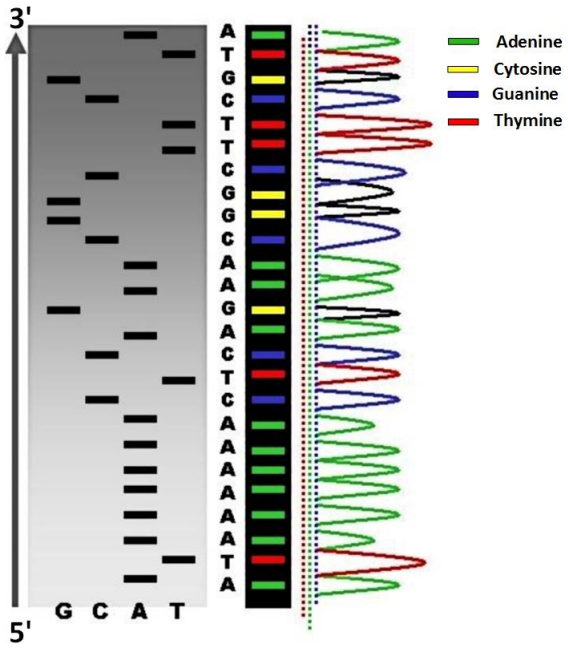

Figure 1.2: Comparison of Sanger methods - gel-electrophoresis ladder (left) and florescent labels (right) [53]. The arrow shows the direction of DNA sequence from 5' end to 3' end.

application to various genomics projects. Several new methods for DNA sequencing were developed in the mid to late 1990s. These techniques comprise many *next-generation sequencing* (NGS) methods. All of NGS techniques achieve high throughput by simultaneously sequencing many DNA sequence templates.

1.1.2 Next generation sequencing

NGS techniques are inexpensive and produce an enormous amount of sequencing data in a short amount of time. The data produced by NGS is much cheaper and faster to obtain than the Sanger reads, however the data contain more errors. Some of the NGS platforms are Roche 454, Ion Torrent and Illumina. The most popular NGS platform is Illumina, which has dominated sequencing space for the last decade. It can sequence small fragments of DNA, called *reads*, that are between 100 to 300 bp long. The details of current Illumina platforms are listed in Table 1.1.

Table 1.1: Output information for Illumina large scale sequencing platforms.

Sequencing Platform	Maximum Read Length	Maximum Reads Per Run	Maximum Output
NextSeq	2 x 150 bp	400 million	120 Gb
HiSeq 2000	2 x 100 bp	2 billion	200 Gb
HiSeq 2500 (High output mode)	2 x 125 bp	4 billion	1000 Gb
HiSeq 2500 (Rapid run mode)	2 x 250 bp	600 million	300 Gb
HiSeq 3000	2 x 150 bp	2.5 billion	750 Gb
HiSeq 4000	2 x 150 bp	5 billion	1500 Gb
HiSeqX Ten	2 x 150 bp	6 billion	1800 Gb
NovaSeq	2 x 150 bp	20 billion	6000 Gb

The Illumina sequencing technology can be summarized in three core steps - amplify, sequence and analyze. The process begins by breaking up a DNA sequence into smaller fragments. These fragments are attached with adapters. An adapter is the oligos bound to the 5' and 3' end of each DNA fragment which act as a reference points throughout the rest of the process. The modified DNA is immobilized on a flow cell surface, called *cluster station*, which facilitates access to enzymes while ensuring stability of surface bound DNA templates. The DNA is then amplified by adding unlabeled nucleotides through a process called *bridge amplification*. During this process, the enzymes incorporate nucleotides to build double-stranded bridges on the surface and then make copies of it. Several million copies of double stranded DNA are generated in each cluster of the flow cell. Primers and flourescently labeled terminators are added to the flow cell that allow primers to add only one nucleotide at a time. A camera is used to take a picture of the cell and a computer determines the base by the wavelength of the fluorescent tag. The process of Illumina sequencing is described in Figure 1.3.

One of the biggest drawback of Illumina technology is the short read length produced by it. Illumina sequencing platform also suffers from biases. A *bias* in DNA sequencing technology is defined as the deviation from the ideal uniform distribution of reads. Both, the short reads and bias limit the application of Illumina sequencing data in downstream applications.

Recently, the focus has shifted from NGS technologies to *Third Generation Sequencing*

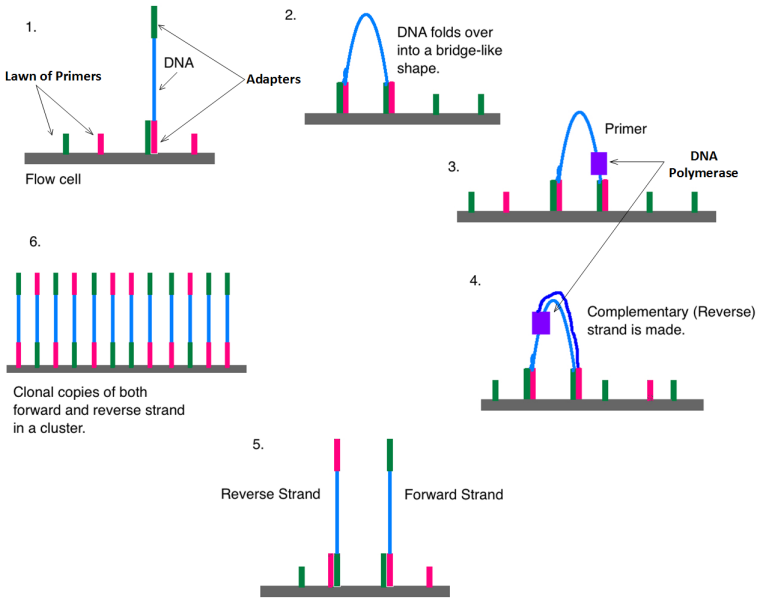

Figure 1.3: Bridge amplification of DNA fragments in Illumina technologies [41].

technologies. One of the most promising third generation sequencing technology is Pacific Biosciences SMRT *(Single Molecule, Real Time)* sequencing. The SMRT sequencing technology produces very long reads ($> 60Kbp$) and uniform coverage across the genome but has error rates even higher than NGS technologies.

1.1.3 Third generation sequencing

SMRT relies on sequencing by synthesis approach and real time detection of incorporated fluorescently labeled nucleotides. It is a parallelized single molecule DNA sequencing method. The two core elements of SMRT sequencing are *zero-mode waveguides* (ZMWs) and *phospholinked nucleotides*. The process begins by affixing a single DNA molecule and DNA polymerase at the bottom of the ZMW as shown in Figure 1.4 (A). ZMWs allow light to illuminate only at the bottom of the well as a DNA molecule is incorporated. Phospholinked nucleotides allow observation of the DNA strand as it is produced. The four bases are tagged with different florescent dyes and are attached to the phosphate chain of the nucleotide. When the DNA molecule is incorporated, the florescent dye is cleaved off from the phosphate chain. A detec-

tor detects the fluorescent signal and the base call is made according to the color of the dye. Figure1.4 (B), shows the details of the incorporation process.

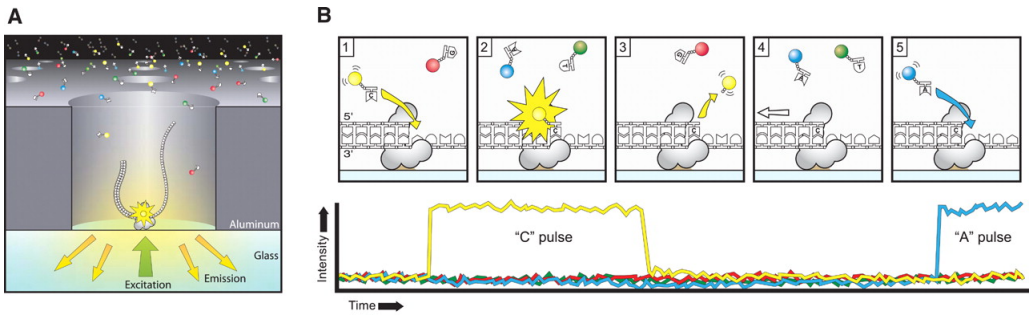

Figure 1.4: (A) ZMW containing template and polymerase. (B) Event sequence of DNA incorporation [20].

The Pacific Biosciences SMRT technology offers long reads with relatively higher error rate ($15 - 20\%$) compared to NGS technologies. The technology is free from biases due to GC rich regions of the genome [49]. For NGS, the coverage levels drop significantly in GC rich regions which makes it impossible to reconstruct these regions of the genome. Since these biases do not exist in SMRT technology, it ensures the uniform coverage of entire genome. *Coverage* is defined as the average number of times each base pair in a genome is sequenced. Given a dataset of n reads of length l with a genome length of L, the coverage is defined as:

$$Coverage = \frac{n \times l}{L}$$

Unlike NGS, the single molecule technology does not require DNA amplification and therefore sampling related biases are also not present. A coverage plot against GC% was plotted for *A.thaliana* genome and shown in Figure 1.5.

The sampling of reads and the errors in reads are completely random. Hence, with sufficient coverage, the effect of high error rate can be mitigated. Figure 1.6 shows a plot between phred quality value and amount of coverage. Phred quality value (QV) is defined as $Q = -10 \log_{10} P$, where P is the base calling error probabilities. Clearly, it can be seen that as the coverage increases, higher accuracy can be achieved. For this example, 80x coverage results in perfect

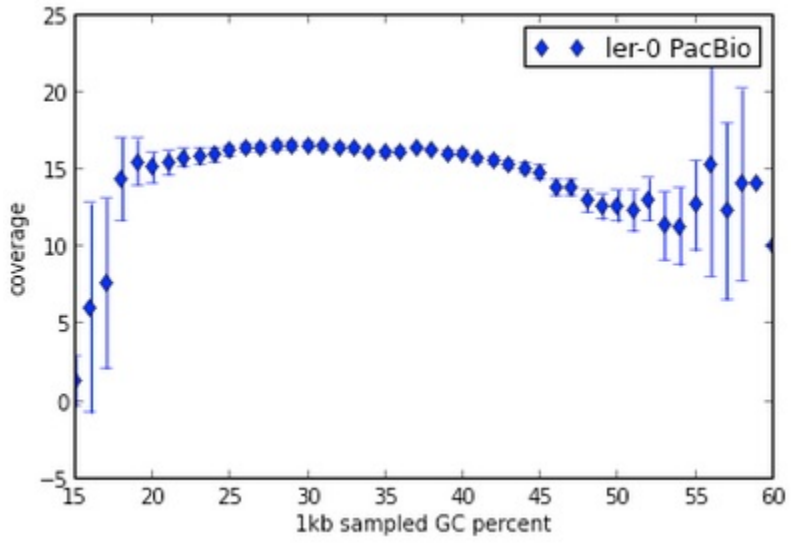

Figure 1.5: Pacific Biosciences unbiased coverage [9].

consensus. A *consensus* sequence is a DNA sequence which is used to describe a number of related but non identical sequences. A *perfect consensus* sequencing data has 100% coverage of nucleotides in reference genome.

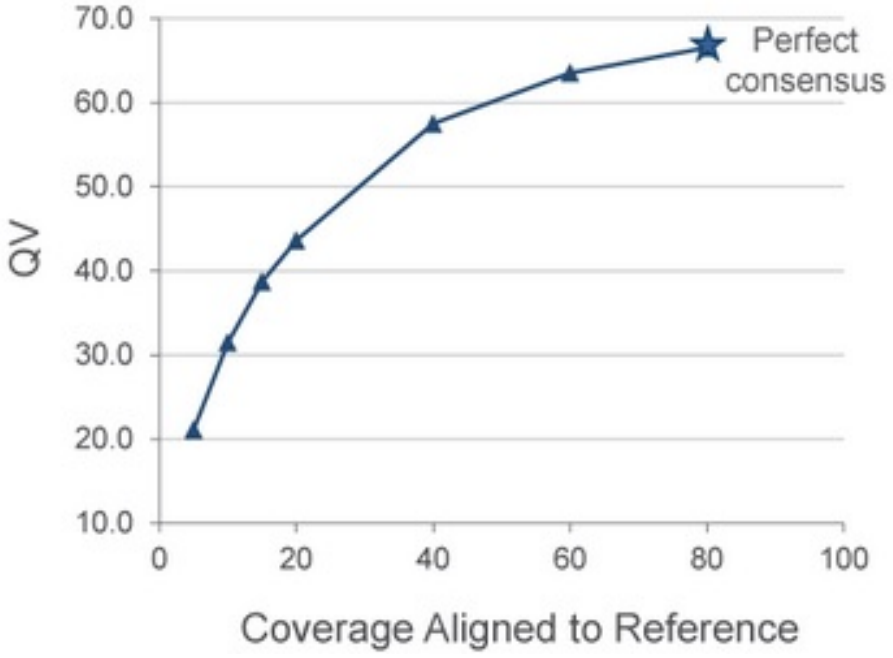

Figure 1.6: Pacific Biosciences Consensus Accuracy [8]

1.2 Maximal Exact Matches

In the remaining part of the introduction, we describe the problems that we are interested in this thesis. In this section, we describe the Maximal Exact Matches (MEMs) computation problem followed by two other sections where we describe the assembly evaluation and sequence alignment problems. MEMs play an important role in genome alignments and comparisons when sequences are relatively similar. MEMs act as seeds in the alignment of high-throughput sequencing reads and are used as anchor points in genome-genome comparisons. Recently, MEMs have been used in Jabba [42] for error correction of long reads obtained from third generation sequencing platforms like Pacific Biosciences and Oxford Nanopore. MEMs are exact matches between two sequences that cannot be extended in either direction without allowing for a mismatch. A related concept with an additional constraint of having a single unique copy in each sequence, called maximal unique matches (MUMs), is also used widely for the same purpose. There are two popular algorithmic approaches have been used in past for computation of MEMs. In the first approach, a compressed index structure of concatenation of both sequences is created and MEMs are computed by iterating over the index structure. This approach has higher memory requirement as both the sequences are used for index creation. In the second approach, the index is created for one of the sequences (reference), and MEMs are computed by matching the second sequence (query) against the index. The second approach has obvious advantages over the first approach in terms of size and re-usability of the index structure.

The E-MEM [30] algorithm is based on the second approach discussed above, where it creates an index for the reference sequence and then computes MEMs by matching and extending k-mers from the query sequence. A k-mer is defined as a small sequence of k characters in the DNA sequence. E-MEM surpasses the state-of-the-art in MEM computation. The details of E-MEM algorithm and techniques are discussed in Chapter 2.

1.3 Assembly Evaluation

Genome assembly is one of the most fundamental problems in Bioinformatics, with many applications. A *genome assembler* is a program which is used to construct the original DNA (or genome) sequence from the reads produced by DNA sequencing. The degree of accuracy in recreating the original DNA from fragments (reads) depends on the error rate, coverage and the relationship between the length of reads and the length of repeats in the DNA being assembled. Assuming low error rate and sufficient coverage, the solution largely depends on the lengths of reads and repeats. A *repeat* is a pattern of DNA sequences which occur in multiple copies throughout the genome. If all repeats are shorter than read length, the solution is trivial. If some repeats are larger than read length, the solution requires trying an exponential number of arrangements which can be computationally very expensive. If most of the repeats are larger than read length, it might be impossible to reconstruct the genome even after trying an exponential number of arrangements. The coverage plays an equally important role, no coverage or very low coverage can adversely affect the accuracy and reconstruction of the genome. Based on the available sequencing technologies, most of the full genome sequences cannot be reconstructed efficiently and reliably. Rather, a fragmented and generally error-prone assembly is produced. Numerous algorithms have been developed and newer algorithms are proposed all the time, trying to achieve accurate genome assembly and longer contigs. A contig is a set of overlapping DNA segments that together represent a consensus region of DNA. It is important to evaluate the quality of produced assembly before being used in further research.

The LASER [29] program is a highly efficient version of the QUAST [25] program, which is more than five times faster and requires only half the memory for large genomes. LASER replaces MUMmer [34, 19, 18] with E-MEM for the MEM computation module in QUAST and makes other changes which result in significant performance improvements. The details of the changes for the LASER program are discussed in Chapter 3.

1.4 Sequence Alignment

Genome or sequence alignment algorithms have been around for more than half a century now. These algorithms have been improved over the years to enhance performance. Figure 1.7 shows a simple example of genome alignment between sequences GAACTA and TAGAA. The gaps or mismatches in the alignment are shown with an underscore character.

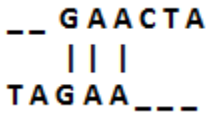

Figure 1.7: Alignment example.

In the last decade, many new algorithms have been developed which are specific to a given sequencing technology. In particular, we are interested in alignment algorithms developed for Pacific Biosciences SMRT sequencing technology. This is a challenging problem because none of the previously developed approaches work well with SMRT data. The SMRT sequencing data suffers from very high error rate, most of the errors being *indels*. Indels refers to an insertion or deletion of nucleotides in the genome of an organism. Further, long read length also poses a significant challenge in maintaining good performance.

The HISEA [31] algorithm is developed in an effort to improve the currently available long read alignment algorithms. An alignment is a way of arranging the DNA sequences to identify regions of similarity which exists due to evolutionary relationships between the sequences. The goal was to develop an aligner with a very high sensitivity and specificity. *Sensitivity* and *Specificity* are statistical measures which compute true positive rate and true negative rate in a given sample of data respectively. The HISEA program consists of many newly developed algorithms which contribute in making it the most sensitive aligner among the other state-of-the-art long read aligners. HISEA is integrated into the Canu assembly pipeline which produces better assemblies than default the Canu pipeline [33]. This validates our hypothesis that a more sensitive aligner can improve the quality of assembly produced by the genome

assembly pipelines for long read sequencing data. The details of the algorithms and techniques for HISEA are discussed in Chapter 4.

Chapter 2

Maximal Exact Matches: E-MEM

2.1 Background

The work outlined here is based on our publication of a new and efficient algorithm for computing *Maximal Exact Matches*, called E-MEM [30]. E-MEM computes all MEMs larger than a given minimum length between two sets of genome sequences. The first set is called a reference sequence and the second is called a query sequence. E-MEM is the best available MEMs computation program in terms of performance. Our program beats all competition with a large margin especially when it comes to large genomes.

2.1.1 Basic Notions and Definitions

Let $\Sigma = \{\$, A, C, G, T, N\}$ be a finite alphabet of ordered characters and let Σ^* be the set of all strings over Σ, including the empty string ϵ. The lexicographical order of characters is $\$ < A < C < G < T < N$, where 'N' represents an ambiguous base and '$' is a sentinel character. Let S is a string of length n over Σ which is always terminated by the character \$. For $0 \leq i < n$, $S[i]$ denotes the character at position i in S. For $i \leq j$, $S[i..j]$ denotes the substring of S starting with the character at position i and ending with the character at position j. The substring $S[i..j]$ is also denoted by the pair (i, j) of positions.

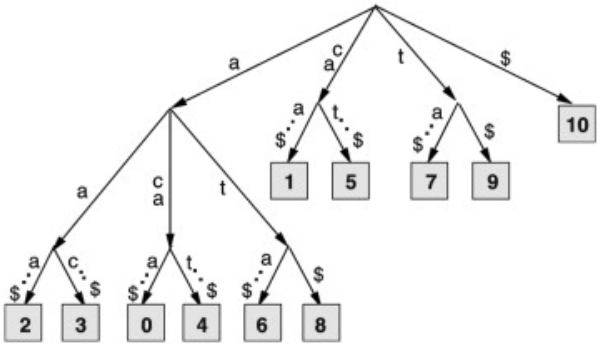

Figure 2.1: The suffix tree for $S = acaaacatat$ [2].

Suffix tree and suffix array

Given a string S of n characters, a *suffix tree* for the string S is a rooted directed tree with exactly $n + 1$ leaves numbered 0 to n. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of $S\$$. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the string S_i, where $S_i = S[i...n-1]\$$ denotes the $i-$th nonempty suffix of the string $S\$$, $0 \leq i \leq n$. Figure 2.1 shows the suffix tree for the string $S = acaaacatat$.

The *suffix array* SA of the string S is an array of integers in the range 0 to n, specifying the lexicographic ordering of the $n+1$ suffixes of the string $S\$$. That is, $S_{SA[0]}, S_{SA[1]}, ..., S_{SA[n]}$ is the sequence of suffixes of $S\$$ in ascending lexicographic order as shown in Table 2.3. The suffix array requires *4n (8n on 64-bit)* bytes of memory hence it is not very memory efficient and cannot be used for large sequences.

The lcp-table LCP (Longest Common Prefix) is an array of integers in the range 0 to n. We define $LCP[0] = 0$ and $LCP[i]$ to be the length of the longest common prefix of $S_{SA[i-1]}$ and $S_{SA[i]}$, for $0 \leq i \leq n$. The lcp-table can be computed as a by-product during the construction of the suffix array or alternatively, in linear time from the suffix array. The lcp-table requires *4n (8n on 64-bit)* bytes in the worst case.

The *inverse suffix array ISA* is a table of size $n + 1$ such that $ISA[SA[q]] = q$ for any

Table 2.1: FM-index of the string $S = acaaacatat$

Occurrence table											
BWT	t	c	a	\$	a	c	t	a	a	a	a
i	0	1	2	3	4	5	6	7	8	9	10
\$	0	0	0	1	1	1	1	1	1	1	1
a	0	0	1	1	2	2	2	3	4	5	6
c	0	1	1	1	1	2	2	2	2	2	2
t	1	1	1	1	1	1	2	2	2	2	2

$0 \leq q \leq n$. *ISA* can be computed in linear time from the suffix array and needs $4n$ bytes.

The table BWT [15] contains the *Burrows and Wheeler transformation* which is a popular algorithm used in data compression. It is a table of size $n + 1$ such that for every i, $0 \leq i \leq n$, $BWT[i] = S[SA[i] - 1]$ if $SA[i] \neq 0$. $BWT[i]$ is undefined if $SA[i] = 0$. The table BWT is stored in n bytes and constructed in one scan over the suffix array in O(n) time.

FM-index

An FM-index [22] is a compressed full-text substring index based on the BWT, with some similarities to the suffix array. It consists of BWT, *Count array* $C[p]$ and *Occurrence table* $Occ(p, k)$. Tables 2.2 and 2.1 show an example of FM-index data structure for string $S = acaaaacatat$.

For each character p in alphabet, Count Array $C[p]$ is defined as the number of occurrences of lexically smaller characters in the string. The table $Occ(p, k)$ is defined as the number of occurrences of character p in $BWT[0..k]$. A major distinction between FM-index and suffix array is the way searching is performed. FM-index searches strings backward while in suffix array string matching is done in forward direction.

Table 2.2: Count array $C[p]$

Count array of *tca*\$*actaaaa*				
p	\$	a	c	t
$C[p]$	0	1	7	9

LCP interval

An interval $[i..j]$, $0 \leq i < j \leq n$, is an *lcp-interval* of lcp-value ℓ if

1. $LCP[i] < \ell$
2. $LCP[k] \geq \ell$ for all k with $i + 1 \leq k \leq j$,
3. $LCP[k] = \ell$ for at least one k with $i + 1 \leq k \leq j$,
4. $LCP[j + 1] < \ell$

A shorthand notation $\ell-$interval (or $\ell - [i..j]$) for an lcp-interval $[i..j]$ of lcp-value ℓ is used. Every index k, $i + 1 \leq k \leq j$, with $LCP[k] = \ell$ is called $\ell-$index. The set of all $\ell-$indices of an $\ell-$interval $[i..j]$ is denoted by $\ell Indices(i, j)$.

2.1.2 MUMmer

MUMmer [34, 19, 18] is a modular and versatile utility that relies on a suffix tree data structure for efficient pattern matching. Suffix trees are suitable for large datasets because they can be constructed and searched in linear time and space. However they have a large memory footprint and that grows rapidly with increasing genome size.

2.1.3 Vmatch

Vmatch [2] is a collection of utilities based on *Enhanced suffix array (ESA)*. The ESA is a data structure consisting of a suffix array and additional tables required to mimic complete suffix

tree behavior. The concept of virtual *LCP-interval tree* was introduced as a part of ESA data structure and it helps to simulate all kinds of suffix tree traversal very efficiently.

The classical method of solving the exact string matching problem using suffix array requires $O(m \log n)$ time, where m is the length of pattern P and n is the length of the text. It was later proved by Manber *et al.* [40] that it can be improved to $O(m + \log n)$ using an additional table. With *ESA*, the exact string matching problem can be solved in $O(m + z)$ time, where z is the number of occurrences of pattern P in the reference string.

DEFINITION

An $m-$interval $[l..r]$ is said to be embedded in an $\ell-$interval $[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leq l < r \leq j$) and $m > \ell$. The $\ell-$interval $[i..j]$ is then called the interval *enclosing* $[l..r]$. If $[i..j]$ encloses $[l..r]$ and there is no interval embedded in $[i..j]$ that also encloses $[l..r]$, then $[l..r]$ is called a *child interval* of $[i..j]$.

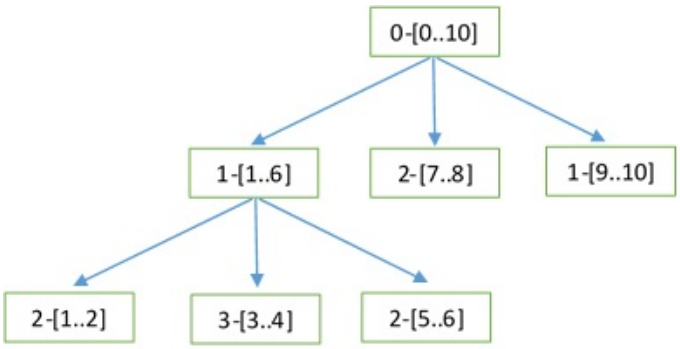

Figure 2.2: The lcp-interval tree of the string S = *acaaacatat* [2].

Based on this definition, a parent-child relationship is created between lcp-intervals, which is called an lcp-interval tree of the suffix array. The root node of this lcp-tree is $0 - [0..n]$. The lcp-interval tree is equivalent to a suffix tree without leaves and there is a one-to-one correspondence between the internal nodes of suffix tree and lcp-interval tree. A leaf of the suffix tree which corresponds to a suffix $S_{SA[l]}$ can also be represented in lcp-interval tree with a *singleton interval* $[l..l]$. Figure 2.2 shows an example of lcp-interval tree of the string S = *acaaacatat*.

The suffix array approach of finding exact matches using binary search requires $O(m \log n)$

time. In order to get the optimal $O(m)$ time complexity, one must be able to determine a child interval in constant time from any lcp-interval $\ell - [i..j]$ instead of $O(\log n)$ used for binary search. This is achieved by an additional table called *child table*.

DEFINITION

The *childtab* is a table of size $n + 1$ indexed from 0 to n and each entry contains three values: *up, down,* and *nextℓIndex*. Formally, the values are defined as follows:

- $childtab[i].up = \min\{q \in [0..i-1] \mid LCP[q] > LCP[i] \text{ and } \forall k \in [q+1..i-1] : LCP[k] \geq LCP[q]\}$
- $childtab[i].down = max\{q \in [i+1..n] \mid LCP[q] > LCP[i] \text{ and } \forall k \in [i+1..q-1] : LCP[k] > LCP[q]\}$
- $childtab[i].next\ell Index = min\{q \in [i+1..n] \mid LCP[q] = LCP[i] \text{ and } \forall k \in [i+1..q-1] : LCP[k] > LCP[i]\}$

The child table stores the parent-child relationship of lcp-intervals. If $\ell - [i..j]$ is an $\ell-$interval and $i_1 < i_2 < ... < i_k$ are the $\ell-$indices in ascending order, then child intervals of $[i..j]$ are $[i..i_1 - 1]$, $[i_1..i_2 - 1]$,...,$[i_k..j]$. Table 2.3 shows the child table of the string $S = acaaacatat$.

Now, we have to simulate suffix links using suffix arrays so that string matching is performed in an efficient manner. It is a property of suffix trees that for any internal node v in the tree with label $\overline{a\omega}$, where $\overline{a}$ is a character and $\overline{\omega}$ is a non-empty string, there exists an internal node u with label $\overline{\omega}$. A pointer from node v to node u is called a *suffix link* of v.

DEFINITION

Let $S_{SA[i]} = a\omega$. If index j, $0 \leq j < n$, satisfies $S_{SA[j]} = \omega$, then we denote j by $link[i]$ and call it the suffix link (index) of i.

The suffix link of i can be computed with the help of the *ISA* as follows

$$link[i] = ISA[SA[i] + 1].$$

Note that the *suffix link interval* obtained using *ISA* and *SA* may not correspond to the actual lcp-interval. Given $\ell - [i..j]$, the smallest lcp-interval $[l..r]$ satisfying the inequality $l \leq link[i] < link[j] \leq r$ is called the suffix link interval of $[i..j]$. The suffix link intervals are stored at the first $\ell-$index position of interval $\ell - [i..j]$ and lcp value of suffix link interval is $\ell - 1$. Table 2.3 shows an example of suffix link interval of string $S = acaaacatat$.

Table 2.3: Suffix array of the string $S = acaaacatat$ with LCP: Longest Common Prefix, BWT: BurrowsWheeler transform, SA: Suffix Array, ISA: Inverse Suffix Array, child table and suffix link tables.

						childtab			**suflink**	
i	$S_{SA[i]}$	BWT	SA	LCP	ISA	up	$down$	$next\ell Index$	l	r
0	$	t	10	0	3					
1	aaacatat$	c	2	0	7		3	7		
2	aacatat$	a	3	2	1				1	6
3	acaaacatat$		0	1	2	2	4	5	0	10
4	acatat$	a	4	3	4				7	8
5	atat$	c	6	1	8	4	6			
6	at$	t	8	2	5				9	10
7	caaacatat$	a	1	0	9	3	8	9		
8	catat$	a	5	2	6				1	6
9	tat$	a	7	0	10	8	10			
10	t$	a	9	1	0				0	10

Computation of MEMs using Enhanced Suffix Array

One approach of computing MEMs is to create an *ESA* of concatenation of sequences X and Y of length m and n respectively. However, this approach is not space efficient. A space efficient approach is to create *ESA* of X and then MEMs are computed by matching suffixes of Y against the *ESA* of X. The following steps describe the MEMs computing algorithm (more details can be found in [2]). A *prefix* of a string is a substring that occurs at the beginning of a string. A suffix of a string is a substring that occurs at the end of a string. A left maximal match is the match which cannot be extended in left of each sequence without a mismatch. Similarly, a right maximal match cannot be extended to the right without a mismatch.

1. Create an *ESA* of string X.

2. Match all suffixes of Y against ESA of X , starting with the longest suffix.

3. Find lower and upper bounds (i.e. LCP intervals of X say $[l..r]$) of a longest prefix match of suffix Y. All the prefixes found are right maximal.

4. Do a depth first traversal (on the virtual LCP interval tree) of all right maximal LCP intervals found in the previous step. Each time a leaf node is encountered, check if $X[i-1] \neq Y[j-1]$, where $i = SA[l']$, l' is the singleton lcp-interval corresponding to the leaf node and j is the current suffix position in Y.

5. The match found is longer than minimum MEM length, report it as MEM.

6. Continue with next suffix of Y.

The MEMs computing algorithm using ESA requires much less memory compared to suffix tree implementation in MUMmer. Later, newer approaches were developed to reduce memory either by compromising runtime performance or by adopting completely new data structures and algorithms. The approaches are discussed in the following sections.

2.1.4 SparseMEM

Table 2.4: Sparse suffix array of the string $S = acaaacatat$ with LCP, and ISA

i	$S_{SA[i]}$	SA	LCP	ISA
0	\$	10	-1	2
1	aaacatat\$	2	0	1
2	acaaacatat\$	0	1	3
3	acatat\$	4	3	4
4	atat\$	6	1	5
5	at\$	8	2	0

In order to reduce the memory footprint of algorithms based on *ESA*, an approach based on *sparse suffix array* was developed [28]. The *sparse suffix array* is a text index based on every K^{th} suffix ($K = 1, 2, ..., n$) of a string. For $K = 1$, the sparse suffix array acts as a full index based on suffix array.

SparseMEM [28] is a MEMs computing program based on this approach. Unlike *Vmatch*, *SparseMEM* does not pre-compute suffix link information. Instead, it is computed whenever required. The suffix link can be computed with the help of the inverse suffix array as follows:

$$l = ISA[SA[l]/K + 1] \text{ and } r = ISA[SA[r]/K + 1]$$

where l and r are the left and right bounds of the suffix link interval respectively. Note that the suffix link interval may require to be extended in both directions for correct results [3]. The *ISA* computation requires minor adjustments to account for sparse suffix index. *ISA* is computed as follows:

$$ISA[SA[j]/K] = j \text{ where } j =, ..., n/K - 1$$

An example of sparse suffix array and corresponding *LCP* and *ISA* values is shown in Table 2.4.

Computation of MEMs using Sparse Suffix Array

Consider two strings *S1* and *S2* and suppose L is the minimum length of MEMs. The following steps describe the MEMs computing algorithm (more details can be found in [28]):

1. Create a sparse suffix array of string *S1*.

2. For any position p in string *S2*, find two lcp intervals using binary search. The first lcp interval $d : [s..e]$ is found by matching at most $L - (K - 1)$ characters while the second interval $q : [l..r]$ is found by matching as many characters as possible i.e. longest possible match. For above two intervals, d and q are the length of the matches, s and l are the start

positions and e and r are the end positions of lcp-intervals respectively. Since $q : [l..r]$ is a subinterval of $d : [s..e]$, therefore $s \leq l$, $r \leq e$ and $q \geq d$ holds.

3. The right maximal matches are found by un-matching characters from interval $q : [l..r]$.

   (a) The first right maximal match is interval $q : [l..r]$.

   (b) The next interval is the parent lcp-interval of $q : [l..r]$. Since $LCP[l] < q$ and $LCP(r+1) < q$, the next lcp-value $q' = max\,(LCP[l], LCP[r+1])$. The boundary position of the parent interval is obtained by extending interval to the left i.e. $l = l - 1$ until $LCP[l] < q'$ and to the right i.e. $r = r + 1$ until $LCP[r+1] < q'$.

   (c) The expansion continues till $q' \geq d$.

4. The right maximal matches found in the previous step are now checked for left maximality. Since the index is sparse by a factor of K, all the left maximal matches are found by scanning upto K characters to the left of the right maximal matches.

5. Advance to the next suffix of *S2* and continue the matching process.

In the last step of the algorithm, suffix link can be used to find the initial lcp-intervals for the next suffix position. This is referred as *suffix link acceleration* in SparseMEM algorithm.

Parallelization technique in sparseMEM

The suffix link acceleration works best with smaller values of K. Note that for an index based on sparse suffix array, moving one suffix position is equivalent to moving K character positions in the string. In step 5 of the previous algorithm, suffix link can only be used if matching for all 0 to $K - 1$ positions in string *S2* is complete. Due to this limitation, improvement from suffix link acceleration starts to diminish for larger values of K.

The performance drop due to increase in sparseness factor is addressed by introducing the concept of parallelization. SparseMEM uses an obvious approach for parallelization which comes directly from the limitation of suffix link application. For all 0 to $K - 1$ positions in

string *S2*, a process is spawned on a separate processor. The results from these K processes are combined before suffix link acceleration is used for moving to the next suffix position.

2.1.5 EssaMEM

An enhanced version of sparse suffix array data structure which includes *sparse child array* is called *Enhanced Sparse Suffix Array (ESSA)*. EssaMEM [59] is a MEM computation program based on *ESSA* which is faster in practice than sparseMEM with the same memory footprint. The essaMEM program proposes two major improvements over the sparseMEM program discussed in the previous section. They are:

1. The addition of sparse child array which allows the traversal of virtual sparse suffix tree in constant time.

2. A new *skip* parameter s, which introduces sparseness in query sequence.

The construction of sparse child array remains the same as discussed previously. Typically, any compressed SA based approach for computing MEMs involves two phases - finding right maximal matches and then extending these right maximal matches to the left for finding left maximal matches. The second phase is usually much faster than the first phase. Based on this observation, the *skip* parameter s is introduced, that increases the work of second phase while decreases the work of first phase by a factor of s. This is achieved by finding right maximal matches of minimum length $L - s.K + 1$ characters, which increases the number of right maximal suffixes. The left maximality is checked for $s.K$ characters to ensure that all the MEMs are larger than length L.

The optimized value of *skip* parameter can greatly improve the performance of MEMs computation algorithm. For optimizing s, the runtime for a maximum of five successive values of s is taken [59]. The largest value of s is set to be the value for which $L - s.K + 1 \geq 10$.

The *skip* parameter does not work well with simulated sparse suffix links, which are suffix links for *ESSA*. Combination of the two requires a mechanism of controlling sparseness factor K for suffix links along with *skip* parameter s of query sequence. This has not been tried out and therefore when suffix links are used, the *skip* parameter s is set to 1.

The MEMs computation algorithm for essaMEM is same as that of sparseMEM, with the two enhancements discussed above.

2.1.6 BackwardMEM

The algorithms discussed so far used indexing techniques to search strings in forward direction. The backwardMEM [46] algorithm uses a data structure which allows searching in backward direction. The data structure used for indexing in backwardMEM is called *FM-index* and it is discussed in section 2.1.1. A typical backwardSearch algorithm is shown in Algorithm 2.1.1. The interval $[i..j]$ corresponds to the currently matched string in suffix array. The character p is the next character in backward search and the new interval is obtained using backwardSearch algorithm.

Algorithm 2.1.1: BACKWARDSEARCH($p, [i..j]$)

$i \leftarrow C[p] + Occ(p, i-1)$

$j \leftarrow C[p] + Occ(p, j) - 1$

if $i \leq j$

  then return ($[i..j]$)

  else return ($NULL$)

Computing Parent Intervals

The *backwardSearch* algorithm generates right maximal matches for a given position in query string. The right maximal matches found correspond to an lcp-interval in a virtual lcp-interval tree. To find the next right maximal match, backwardMEM program stores two tables, *previous smaller values (PSV)* and *next smaller values (NSV)*.

The PSV and NSV table entries are computed as follows:

$$PSV[i] = max\{k \mid 0 \leq k < i \text{ and } LCP[k] < LCP[i]\}$$
$$NSV[i] = min\{k \mid i < k \leq n \text{ and } LCP[k] < LCP[i]\}$$

Once the PSV and NSV information is stored as part of the data structure, the parent interval for an lcp-interval $[i..j]$ with $LCP[i] = p$ and $LCP[j+1] = q$ is determined as:

$$parent([i..j]) = \begin{cases} p - [PSV[i]..NSV[i] - 1 & \text{if } p \geq q \\ q - [PSV[j+1]..NSV[j+1] - 1] & \text{if } p < q \end{cases}$$

Table 2.5 shows an example of the PSV and NSV values for string $S = acaaacatat$.

Table 2.5: PSV and NSV tables of the string $S = acaaacatat$.

i	$S_{SA[i]}$	BWT	SA	LCP	PSV	NSV
0	$	t	10	0		
1	aaacatat$	c	2	0		
2	aacatat$	a	3	2	1	3
3	acaaacatat$		0	1	1	7
4	acatat$	a	4	3	3	5
5	atat$	c	6	1	1	7
6	at$	t	8	2	5	7
7	caaacatat$	a	1	0		
8	catat$	a	5	2	7	9
9	tat$	a	7	0		
10	t$	a	9	1	9	

Computation of MEMs using the FM-index

Consider two strings *S1* and *S2* and suppose L is the minimum length of MEMs. The following steps describe the MEMs computing algorithm:

1. Create an FM-index data structure of string *S1*.

2. Start with the right most character of string *S2*. Use *backwardSearch* algorithm for finding right maximal matches of length $\geq L$. The right maximal match is a triplet $(q, [l..r], p'_2)$, where q is the length of the match, $[l..r]$ is the left and right bounds in suffix array and p'_2 is the position in *S2* for this match. The current match in string *S2* is indicated by the length $p'_2 + q - 1$.

3. For each triplet $(q, [l..r], p'_2)$ and $q \geq L$,

   (a) The left maximality is checked by checking $BWT[k] \neq S2[p'_2 - 1]$, where $l \leq k \leq r$.

(b) If left maximal, report MEM.

(c) Find parent interval of $(q, [l..r])$. Continue until $q \geq L$ for parent interval.

4. Continue with the current position in string $S2$.

In order to get a smaller memory footprint, the FM-index ($LF-$mapping) is stored in a wavelet tree [60]. A wavelet tree is a data structure which stores sequences in compressed form. Note that it is not necessary to store BWT, which is only required during left maximal comparison. This is because the wavelet tree allows to access $LF-$mapping without it, and we have $BWT[k] \neq p$ if and only if $LF(k) \notin [i..j]$, where $[i..j]$ is the current $p-$interval (e.g., *backwardsearch(p, [1..n])* returns $[i..j]$). This is same as replacing the test $BWT[k] \neq S2[p'_2 - 1]$ with the test $LF(k) \notin [i..j]$, where $[i..j]$ is the $S2[p'_2 - 1]-$interval.

Compressed Suffix Array implementation

BackwardMEM also supports a version of program based on *compressed suffix array*. There is a difference between a compressed suffix array and a sparse suffix array - a compressed suffix array stores each k^{th} entry of the suffix array $S1$ while a sparse suffix array stores each K^{th} suffix of $S1$.

The obvious advantage of compressed suffix array is a smaller memory footprint. This is further reduced by storing the compressed suffix array in a wavelet tree which requires only $(n \log n)/k$ bits, where n is length of string $S1$ and k is the compress parameter. The size and access time for compressed suffix array depends on compress parameter $k \geq 1$. For $k = 1$, the compressed suffix array acts as a full suffix array and the access time is constant. For $k > 1$, every k^{th} entry of suffix array is stored and the remaining entries are constructed in $k/2$ steps [46] with $LF-$mapping.

2.1.7 SlaMEM

The MEMs computing program *slaMEM* [21] is an improvement to the data structures used in *BackwardMEM* [46]. There is no difference in MEMs computing algorithm as far as the logical steps are concerned. Instead, new remodeled data structures have been proposed for improving runtime performance and reducing the memory footprint. The improvements are based on following observations:

1. LCP array is accessed most frequently for resolving parent intervals. A fast mechanism of retrieving lcp-values helps in improving the performance.

2. To compute the parent intervals, only boundary lcp-values are needed. Having only boundary lcp-values reduces the memory footprint.

Sampled LCP Array

Based on the above observations, *slaMEM* samples only boundary values for the LCP array. The PSV and NSV arrays are also computed for these sampled LCP positions. The boundary positions are sampled as follows:

$$TopCorners = \{i : (i+1) \neq n \text{ and } LCP[i] < LCP[i+1]\}$$
$$BottomCorners = \{i : (i+1) = n \text{ or } LCP[i] > LCP[i+1]\}$$

where $TopCorners$ and $BottomCorners$ represent the boundary positions of lcp-interval corresponding to a BWT string.

Sampled Smaller Values

The computation of parent intervals requires only a PSV value corresponding to $TopCorners$ and NSV value corresponding to $BottomCorners$. Therefore, it is sufficient to keep a single *Sampled Smaller Value (SSV)* array instead of both PSV and NSV arrays. The array $SSV[i]$ will hold a value of $PSV[i]$ if the position corresponds to $TopCorners$ or $NSV[i]$ if the position corresponds to $BottomCorners$. This can be represented by following equations:

$$SSV[i'] = PSV[i], \qquad \text{if } SSV[i'] < i$$
$$SSV[i'] = NSV[i+1] - 1, \qquad \text{if } SSV[i'] > i$$

where i' is the number of sampled positions in the interval $[0, (i-1)]$ because SSV does not have the same size as PSV/NSV.

It is possible to have overlapping left or right interval positions for two or more intervals. In such cases, the $SSV[i]$ will store the left or right most parent interval of the overlapping positions. The missing intervals are found by a scan of closest top or bottom corners around that position. Since the SLCP is sampled, the search is much faster than using full LCP array.

2.1.8 Comparison

Table 2.6 summarizes the important differences and similarities among the applications discussed in this text. The memory requirement of each tool is shown in terms of bytes per character. The K and k are sparseness factor and compress parameter respectively. Once the index is created, each of these algorithms can find MEMs in theoretical time complexity proportional to the length of the pattern.

Table 2.6: A nutshell comparison of applications. Notations used in the table: ST (Suffix Tree); ESA (Enhanced Suffix Array); SSA (Sparse Suffix Array); ESSA (Enhanced Sparse Suffix Array); LCP (Longest Common Prefix); CT (Child Table); BS (Binary Search).

	MUMmer	**Vmatch**	**sparseMEM**	**essaMEM**	**backwardMEM**	**slaMEM**
Text Index	ST	ESA	SSA	ESSA	FM-index	FM-index
Parallelization	Yes	No	Yes	No	No	No
Initial Search	ST	LCP CT	BS	LCP CT	Count & Occ Tab	Count & Occ Tab
Suffix Link	Yes	Yes	Yes	No	No	No
Memory (bytes)	$17n$	$10n$	$(9/K+1)n$	$(9/K+1)n$	$(4/k+2)n$	$2.2n$
Flexibility (Memory vs Time)	No	No	Yes	Yes	Yes	No
Search Order	Forward	Forward	Forward	Forward	Backward	Backward
Commercial	open source	closed source	open source	open source	open source	open source

2.2 E-MEM algorithm

As discussed in previous sections, a typical MEM computation algorithm creates an index of the reference sequence, which is used for quickly finding seed matches. The seeds are then extended to find a possible MEM. Depending on the indexing technique, the extension of seeds may or may not use the index during the extension phase. E-MEM is designed by using an efficient implementation of simple algorithmic ideas. The algorithm creates an index based on double hashing which stores k-mers and its positions in the reference. All query k-mers are matched against this index. The matches are extended while ensuring that any k-mer matches resulting in a previously discovered MEM are discarded. E-MEM does not rely on the index during the extension phase. A high-level overview of the E-MEM algorithm is provided in Algorithm 2.2.1. The algorithm requires three mandatory input parameters - a reference sequence R, a query sequence Q and minimum MEM length $minL$. An optional parameter, division factor (D), is also supported for memory reduction which is discussed

in Section 2.9. The default value of this parameter is set to one, which means no splitting is performed and full genomes are used. The input sequences are required to be in FASTA format. *FASTA* [37] is a text based format for representing DNA sequences in which nucleotides are represented by a single character codes. It also allows the sequence names and comments to precede the sequences. An *exact match* between two sequences R and Q is a triple $(len, s1, s2)$ such that $len \geq minL$, $s1 \in [0, |R| - len]$, $s2 \in [0, |Q| - len]$, and $R[s1..s1 + len - 1] = Q[s2..s2 + len - 1]$. An exact match is *left maximal* if $R[s1 - 1] \neq Q[s2 - 1]$ and right maximal if $R[s1 + len] \neq Q[s2 + len]$. A *maximal exact match (MEM)* is a left and right maximal exact match.

The algorithm starts by encoding and hashing the reference sequence R. The hash table size is kept roughly two times the number of *k-mers* in the sequences. The number of k-mers is estimated based on an approximation which uses total number of base pairs in reference, the *k-mer* size and minimum MEM length *minL* . Next, a query sequence Q is encoded and then iterated over for a possible *k-mer* match in the reference hash. If a query *k-mer* match is found in the hash table for the reference, this *k-mer* is extended, both in left and right direction until a mismatch is encountered. If the length of the match is greater than *minL*, the match is reported as a MEM. The E-MEM program can run in serial and parallel mode. The parallelization is done using OpenMP directives. The algorithm 2.2.1 uses a CurrMEM (linked list) - to keep track of the previously discovered MEMs. The MEMs spanning across D spits are tracked in a vector MEMext which are merged before final MEMs are reported.

Algorithm 2.2.1: E-MEM($R, Q, minL$)

comment: Input - two sequences R and Q and a minimum MEM length minL

Choose a division factor D

Split R into $R_1, R_2, ..., R_D$; Split Q into $Q_1, Q_2, ..., Q_D$

comment: Splitting details discussed in Section 2.9

for $i \leftarrow 1$ **to** D

do
- Encode R_i
- **comment:** Encoding discussed in Section 2.3
- $l \leftarrow minL - k + 1$
- **comment:** k is $k - mer$ size
- **while** ($l <= |R_i| - k + 1$)
  - **do**
    - Hash the $k - mer$ at position l of R_i
    - $l \leftarrow l + minL - k + 1$
- **for** $j \leftarrow 1$ **to** D
  - **do**
    - Encode Q_j
    - **for** $l \leftarrow 1$ **to** $|Q_j| - k + 1$)
      - **do**
        - Get the $k - mer$ q at position l in Q_j
        - Search for $k - mers$ $r = q$ in the hash table of R_i
        - **for** Each occurrence of r with extension >= $minL$
          - **do**
            - Check CurrMEMs
            - **if** ((q, r) discovers a new MEM)
              - **then if** (MEM at ends of R_i or Q_j)
              - **then** Add to MEMext
              - **else** Add to file (by start position in Q_j)
        - Update CurrMEMs
- Remove Hash table for R_i

Process MEMext to extend MEMs

Move MEMs from MEMext to appropriate files

for Each file with MEMs

do
- Sort MEMs by position in Q
- Remove duplicates

Output MEMs

2.3 Sequence Storage

The E-MEM program uses many techniques to reduce memory and improve performance. To reduce memory requirements, the sequences are read from FASTA files and stored in a 2-bit encoding scheme. All four nucleotides can be represented with 2-bits as {A=00, C=01, G=10, T=11}. The storing of a nucleotide in 2-bits instead of a byte reduces memory requirement by a factor of 4. An array of unsigned 64-bit integers is used to store the entire genome with a single unsigned 64-bit holding 32 nucleotides. The genome sequence containing base pair N (an ambiguous base) are replaced with a random nucleotide and the position is tracked for final MEM reporting. The details of unknown base handling are discussed in Section 2.8.

2.4 Efficient *k*-mer Storage

We observed that storing all *k*-mers in the reference sequence is unnecessary as it leads to redundant MEM discovery. To avoid redundant hits, *k*-mers are stored at intervals of length $minL - k + 1$, where *minL* is minimum MEM length and *k* is the *k*-mer size. This ensures that at least one *k*-mer is stored in any possible MEM of length *minL*. Figure 2.3 shows three consecutive positions for *k*-mer hashing. It is clearly seen that storing *k*-mers at an interval reduces memory requirement and processing time for hashing. It also avoids redundant hits during later stages, which results in significant performance improvements.

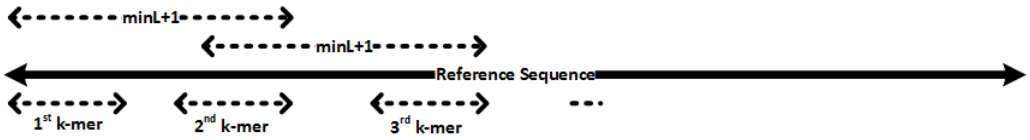

Figure 2.3: *k*-mer hashing technique: only the *k*-mers shown are stored

2.5 Hash Table and hashing function

The E-MEM algorithm uses hashing to efficiently store and search k-mers for matching between query and reference sequences. The technique involves creation of an associate array abstract data structure, called *hash table*. A hash table uses a *hash function* to compute an index which locates the desired value. The average time complexity of *insert, delete* and *search* operation is $O(1)$ which makes hashing a very efficient data structure.

The input to E-MEM hashing function is an unsigned value obtained from 2-bit representation of k-mers. The hash function then computes a modulus of input value with the size of the hash table, which is used as an index in the hash table. The hash table size is a prime number to ensure that all indexes are probed. It is possible that hash function maps two different k-mer values to same index in the hash table. This is called *collision* and over the years many techniques are developed to deal with collision in hashing algorithms. A popular method for collision resolution in hash tables is called *Open Addressing*. In this method, the collisions are resolved by searching through alternate locations in hash table until either the key is found or an unused location is found, which indicates that the key does not exist in the hash table. The three most common open addressing approaches are *Linear Probing, Quadratic Probing* and *Double Hashing*. The objective of any good hashing technique is to distribute the keys evenly to avoid collisions while maintaining constant time performance of basic operations.

E-MEM uses double hashing technique for efficient storage and retrieval of k-mer values. The load factor is kept under 50% to maintain high performance. At each index in the hash table where a k-mer maps, a list of reference position are stored. The positions are maintained in an array which grows in powers of 2. The first position in the array stores the number of reference positions in the array. The array grows dynamically as it becomes full and space is needed for adding new reference positions.

2.6 Searching query

Storing all k-mers in the reference sequence is not necessary, as mentioned in Section 2.4 however all k-mers in the query sequence have to be considered. For every query k-mer, all matching k-mer positions in the reference are investigated for a possible MEM. An efficient implementation is achieved by performing bit level operations. A 64-bit sliding window is used to read query sequence and a bit mask of $2k$ 1's extracts the k-mer bits. Moving the sliding window by 2 bits to the right gives the next k-mer in the query sequence. To maintain an efficient sliding operation, a byte (8 bits) is shifted each time instead of two bits. This restricts the max k-mer size to 28 nucleotides.

Once a k-mer hit between query and reference is found, it needs to be extended efficiently in both directions for a possible MEM. Extending one character at a time will result in a very inefficient algorithm. Since the sequences are stored in blocks of 64 bits, comparisons are performed using very few bit operations as shown in Figure 2.4.

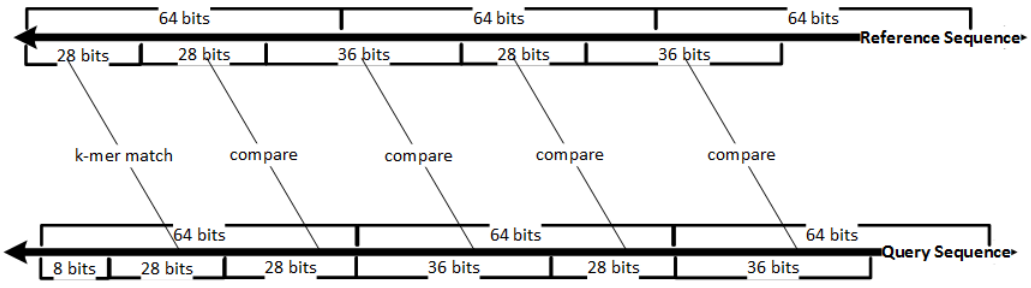

Figure 2.4: Efficient k-mer matching

For example, Figure 2.4 shows a k-mer match of size 14 or 28-bits found between query and reference sequence. Next, the extension is performed by matching the query sequence in 2 blocks of 28 and 36 bits. A 64-bit block is compared using two comparison operations and few bit operations, which makes the entire process very efficient.

2.7 Handling redundant MEM matches

The E-MEM program uses many techniques to make it more efficient. The algorithm extends initial k-mer hits between query and reference sequences to find a MEM. Since every k-mer in the query sequence is looked up in the reference hash, it is expected that a MEM longer than minimum MEM length $minL$ is discovered by at least two k-mers. The second k-mer hit simply rediscovers the previously found MEM. Figure 2.5 shows a possible scenario for one such case. It is important that all such cases are discovered quickly and discarded for efficient functioning of the E-MEM program.

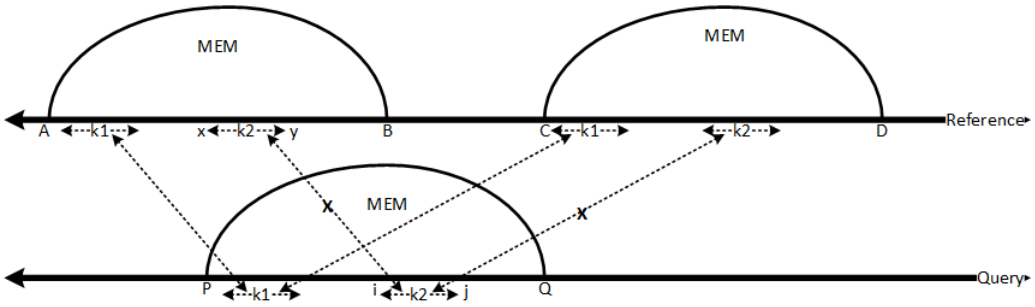

Figure 2.5: Redundant MEMs

The E-MEM algorithm avoids redundant computation of MEMs by keeping track of the relative distance of the current k-mer position with respect to the already discovered MEMs in the query sequence. The relative distance thus obtained is used with the current k-mer position in the reference sequence to compute MEM coordinates. If the computed MEM coordinates match with a previously discovered MEM, the k-mer is discarded, otherwise k-mer is extended for a possible MEM. Discarding k-mers based on above computation results in significant performance improvement. Figure 2.5 has a reference and a query sequence marked $Reference$ and $Query$ respectively. The two different k-mers, $k1$ and $k2$ are shown with bidirectional lines. The semi-elliptical shapes represent the MEMs of length $l > minL$. Two MEMs (A, P, l) and (C, P, l) are discovered by extending k-mer match $k1$ between query and reference where A and C are starting positions of MEMs in the reference sequence and P is the starting position of MEM in the query sequence. The k-mer match $k2$ is embedded in a previously discovered

MEM. Therefore, it is not extended, as shown with a dotted line with a cross on it. To avoid the redundant computation, a check is performed to ensure that extension of this k-mer is not going to produce a previously discovered MEM. The relative distance of $k2$ from left and right side in query sequence is $i - P$ and $Q - j$ respectively, where i and j are start and end positions for $k2$. This information is now used to compute a MEM coordinate in reference i.e. if $x - i + P = A$ and $y + Q - j = B$, then k-mer $k2$ is discarded. Note that only matches of k-mer $k2$ which rediscover an existing MEM are discarded. All other matches of $k2$ are still checked for a possible MEM (not shown in Figure 2.5).

2.8 Dealing with ambiguous bases (N)

E-MEM uses 2 bits to represent four nucleotides which reduces memory requirements. However, genome sequences sometimes have an unknown or ambiguous base represented by the character N. Since E-MEM can store only four bases with 2 bit memory representation, it randomly replaces the ambiguous base with a valid nucleotide. This may result in inaccurate MEM reporting if not handled properly. E-MEM keeps track of all blocks of N's separately for reference and query sequences. This information is used at various stages in E-MEM algorithm.

First, it is used to avoid storing of any k-mers with an ambiguous base during reference hashing. Similarly, when query k-mers are read, this information is again used to ignore all k-mers with an ambiguous base. The above ensures that no k-mer matching is performed with an ambiguous base, as all such k-mers have already been filtered out. However, there is still a possibility of reporting a MEM with an ambiguous base. During the k-mer extension phase, it is possible that the extension matches with an ambiguous base which was randomly replaced during the sequence storing process. This scenario is very unlikely, but still possible. To ensure that no MEMs are reported with an ambiguous base, checks are performed with stored blocks of N's.

2.9 Split parameter - memory reduction

To reduce the memory requirements further, E-MEM also provides an optional division of sequences into smaller parts. The E-MEM program works by loading smaller parts of sequences - one at a time, which reduces memory requirement by a factor of D - the number of divisions, also called the *division factor*. It is not possible to cut the sequences in $D \geq 1$ equal subsequences since this may result in loss of MEMs which span across the cut. To avoid this problem, we keep D sequences overlapping such that each subsequence has a length $l = \frac{1}{D}(|R| + (D-1)minL)$ and an overlap of $minL - 1$. One can clearly see that MEMs spanning across the cut boundary need to be merged before being reported. All MEMs starting or ending at a subsequence border are tracked and merged before reporting.

Theoretically, one can divide sequences into as many smaller parts as possible. However, divisions have a negative impact on performance, therefore more than 10 divisions are not recommended. The program requires extra computation time to merge and process the MEMs which span across these divisions.

2.10 Very large number of MEMs

For large genomes which are highly similar, the number of MEMs can increase drastically. This situation can also arise if minimum MEM length is very small. In such cases, if all the MEMs are stored in memory, this would increase memory requirements by many folds. We observed this behavior when computing MEMs for Human versus Chimp, which resulted in more than 100 million MEMs. To avoid this problem, the MEMs are stored in temporary files, sorted according to the query start position. Once the processing of MEMs is complete, the duplicates are removed from the temporary files and all MEMs are merged in the right order.

2.11 Output formats

The E-MEM program prints MEMs on standard output (*stdout*). The output format varies depending on the command line options used. In the default mode, 3-column output is printed as shown in Figure 2.6. By default, the E-MEM program computes and prints MEMs only in forward direction. It is possible to enable MEM reporting in both forward and reverse complement directions. For each query sequence, the sequence name or ID is reported on the first line followed by a > character. The sequence name will be reported even if there are no MEMs found for this sequence. For example, the query sequence in "Reverse" reports a sequence name with no matching MEMs in Figure 2.6. Note that, for each query sequence, the reverse complemented MEMs immediately follow the forward MEMs. For each match, the 3-columns list the start position in the reference sequence, the start position in the query sequence and the length of the match respectively. For reverse matches, the positions are reported relative to the reverse query sequence.

```
> gi|5524211|gb|AAD44166.1|
     4              1             51
     8              1             51
    12              1             51
    16              1             51
    20              1             51
    24              1             51
    28              1             51
     1              2             50
> gi|5524211|gb|AAD44166.1| Reverse
```

Figure 2.6: Example: 3-column output

The 3-column output is sufficient for reporting matches between one reference sequence and one or more query sequences. For multiple reference sequences, 4-column output is required. E-MEM provides an option -F to print 4-column output which also prints the reference

sequence name or ID for each match. Figure 2.7 shows an example of 4-column output. The first column represents the name or ID of the reference sequence followed by the 3-column output format as described above.

```
> gi|5524211|gb|AAD44166.1|
   ref1                    4               1               51
   ref1                    8               1               51
   ref2                   12               1               51
   ref2                   16               1               51
   ref3                   20               1               51
   ref3                   24               1               51
```

Figure 2.7: Example: 4-column output

2.12 Results

The performance of E-MEM was compared against state-of-the-art software programs from other research groups. The programs used are essaMEM [59], slaMEM [21], sparseMEM [28] and Vmatch [2]. Other programs which were also considered are backwardMEM [46] and MUMmer [34, 19] but dropped from consideration as they fail to run or are very slow with our datasets. The genomes used for testing performance are shown in Table 2.7. Note that the wheat genome is about 17Gbp, however the available sequence is of size 4.3Gbp only. All the programs were first tested manually for correctness using smaller datasets. The correctness is further ensured by computing MEMs for larger datasets and verifying it against output of other programs. Any deviation from majority is considered an error in program output.

2.12.1 Evaluation

Tests were performed in various settings including different minimum MEM length, serial mode and parallel mode. The MEMs were computed between human vs mouse, human vs chimp and common wheat vs durum wheat. Minimum MEM length of 100 and 300 bps were

Table 2.7: Genomes used for testing

Datasets	Size (Mbs)	Number of Sequences
Homo sapiens (Human)	3,137	93
Mus musculus (Mouse)	2,731	66
Pan troglodytes (Chimp)	3,218	24,132
Triticum aestivum (Common wheat)	4,391	731,921
Triticum durum (Durum wheat)	3,229	5,671,204

used to check any bias towards it. All tables showing results use a dash (-) for the programs that could not run that test for some reason. For example, a dash in serial mode result means that output was incorrect or empty, whereas a dash in the parallel mode means that it was not supported by the program.

The competing programs were run according to the specifications in their respective manuals, websites, and readme files. All tests were run on the Shared Hierarchical Academic Research Computing Network (SHARCNET), with a DELL PowerEdge R620 computer with 12 cores Intel Xeon at 2.0GHz and 256GB of RAM, running Linux Red Hat, CentOS 6.3.

2.12.2 Human vs Mouse

The test was conducted with whole human and mouse genomes. Tables A.1 and A.2 show results for minimum MEM length 100 and 300 respectively.

Minimum MEM length 100

For human vs mouse, there are 537,491 MEMs of minimum length 100 reported by each program. In serial mode, the least memory was reported by slaMEM, which was 3.5GB. However, slaMEM took 17 hours to complete the test. The best performance among the competing programs is by essaMEM, which uses 4GB of memory, but takes only 2.5 hours to complete the test. essaMEM was able to complete the test in little over one hour, but the memory was increased enormously to 19GB. E-MEM can run within as little as 623MB and complete in less than two hours or in half an hour using 4GB of memory, which makes it clearly superior with

respect to both parameters of memory and time.

E-MEM has the best performance gains when serial mode is changed to parallel model, between 5 and 6 times on a 12 core machine. The gains in essaMEM varies between 1.3 and 4.5 times. We observed that sparseMEM has a performance gain comparable to that of E-MEM, but its running time in serial mode is way higher. Therefore, even after the gains similar to E-MEM, the difference remains large. Among all competing programs, the best results come from essaMEM, that consistently outperforms the others - 23 min and 6GB or one hour and 5.4 GB. E-MEM can complete the test in 6 minutes and 4.7GB or 10 minutes and less than 2GB. Refer Appendix A for detailed results.

The time and space are plotted in Figure 2.8. Programs closer to the origin are faster and need less memory. The top plot is for serial mode and the bottom is for parallel. The area is very large in the serial plot, due to the large memory requirements of Vmatch and long running times required by slaMEM and sparseMEM. Since the two programs do not run in parallel, the plot giving the time/space values for the parallel testing gives a more clear picture.

Only essaMEM, sparseMEM, and E-MEM are able to run in parallel. The memory required for the same sparseness factor by essaMEM is slightly less than that of sparseMEM. However, essaMEM is much faster than sparseMEM. This is expected because essaMEM program is an improvement of sparseMEM algorithm and it use the same codebase. All three programs trade time for space but the results of E-MEM are much closer to origin.

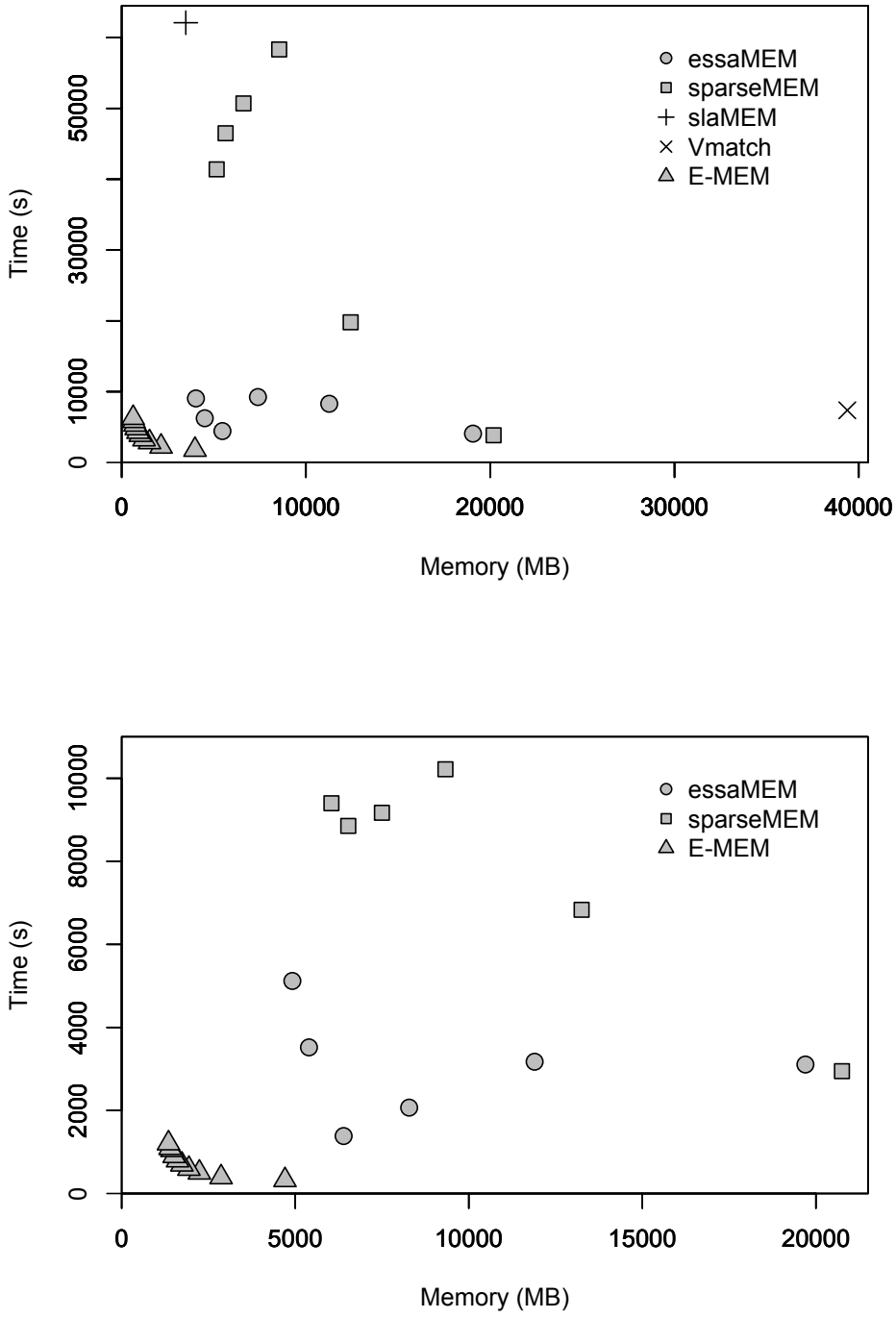

Figure 2.8: *Homo sapiens* vs *Mus musculus*; MEMs of minimum length 100. The top plot is for serial mode, the bottom for parallel. Note the different scale of the plots.

Minimum MEM length 300

Similar to minimum MEM length 100, results for minimum length 300 show that E-MEM has best performance for both serial and parallel mode. There are 390 MEMs of minimum length 300 reported by each program. Figure 2.9 shows time and space plot for all the programs. E-MEM involves a post processing step and its time is dependent on the number of MEMs reported. Since the number of MEMs is much lower compared to MEMs for minimum MEM length 100, the post processing step requires less time and hence there is an improvement in overall performance of E-MEM. Refer Appendix A for detailed results.

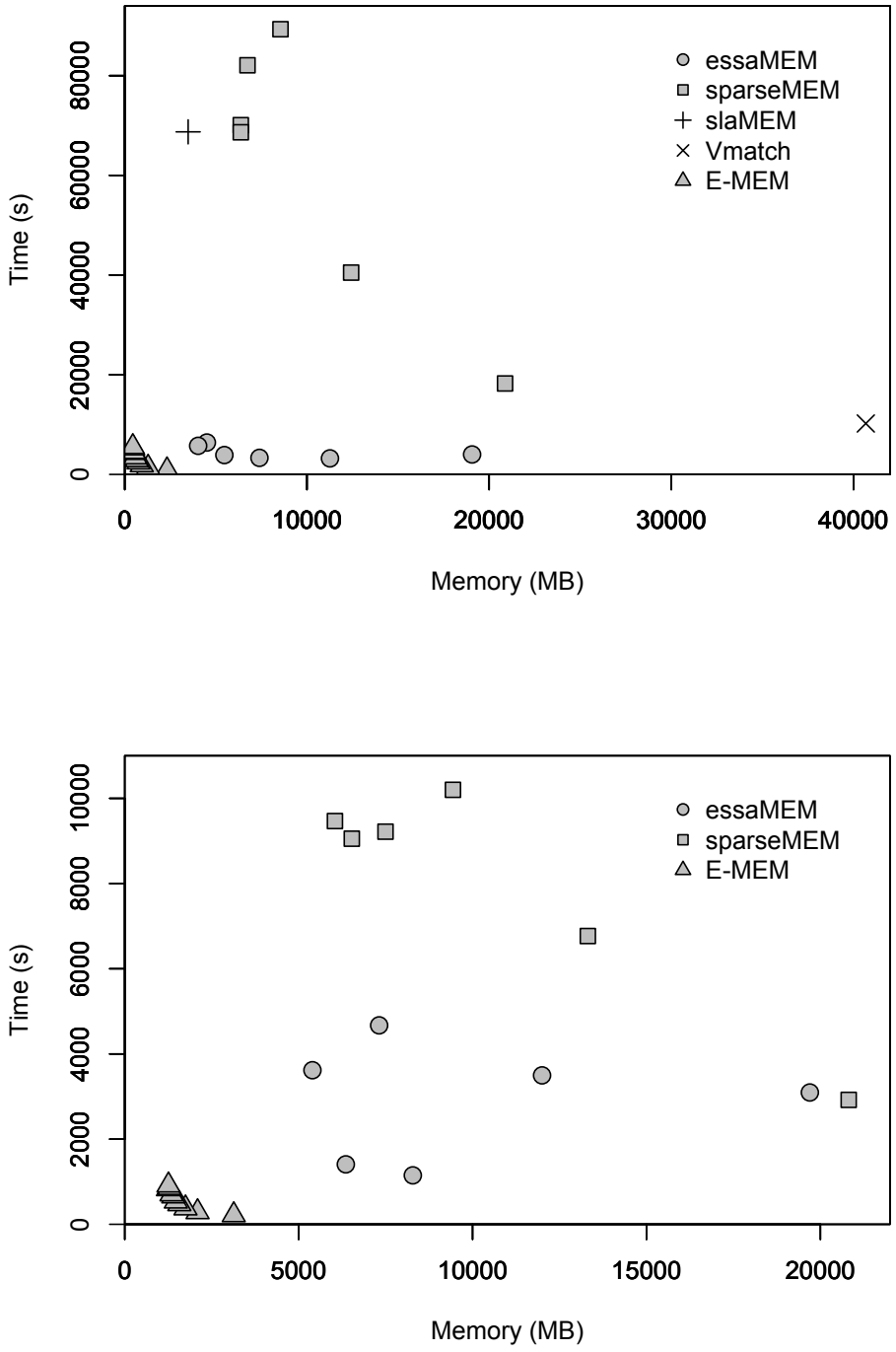

Figure 2.9: *Homo sapiens* vs *Mus musculus*; MEMs of minimum length 300. The top plot is for serial mode, the bottom for parallel. Note the different scale of the plots.

2.12.3 Human vs Chimp

The test was conducted with whole human and chimp genomes. Tables A.3 and A.4 show results for minimum MEM length 100 and 300 respectively.

Minimum MEM length 100

For human vs chimp,132,368,058 MEMs of minimum length 100 are reported by all programs. The performance of E-MEM program slightly diminishes as a result, since all these MEMs need to be post processed. (The very large number of MEMs is also the reason why the program is slightly slower for $D = 1$ as compared to $D = 2$.) The details are given in Table A.3. The whole picture as given in Figure 2.10 is similar with the one for the first test, human vs mouse Figure 2.8 except that the difference between the running times is not as high. The top plot is for serial mode, the bottom for parallel. Refer Appendix A for detailed results.

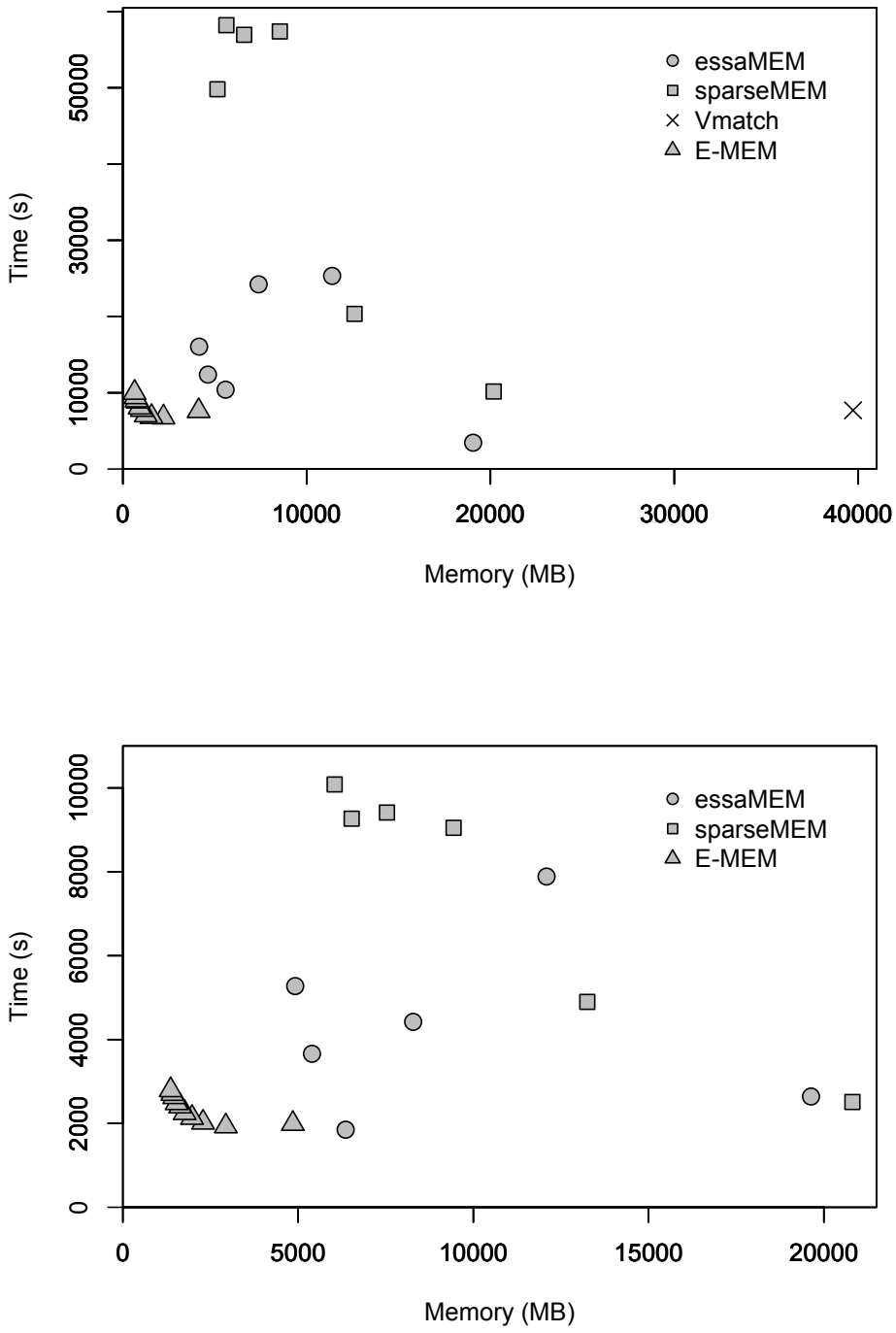

Figure 2.10: *Homo sapiens* vs *Pan troglodytes*; MEMs of minimum length 100. The top plot is for serial mode, the bottom for parallel. Note the different scale of the plots.

Minimum MEM length 300

For minimum MEM length 300 - 951,561 MEMs are reported by all programs. It can be clearly seen that performance of E-MEM is much better than competing programs. The performance of E-MEM for minimum MEM length 300 improves compared to minimum length 100 as the number of MEMs is greatly reduced. Refer Appendix A for detailed results.

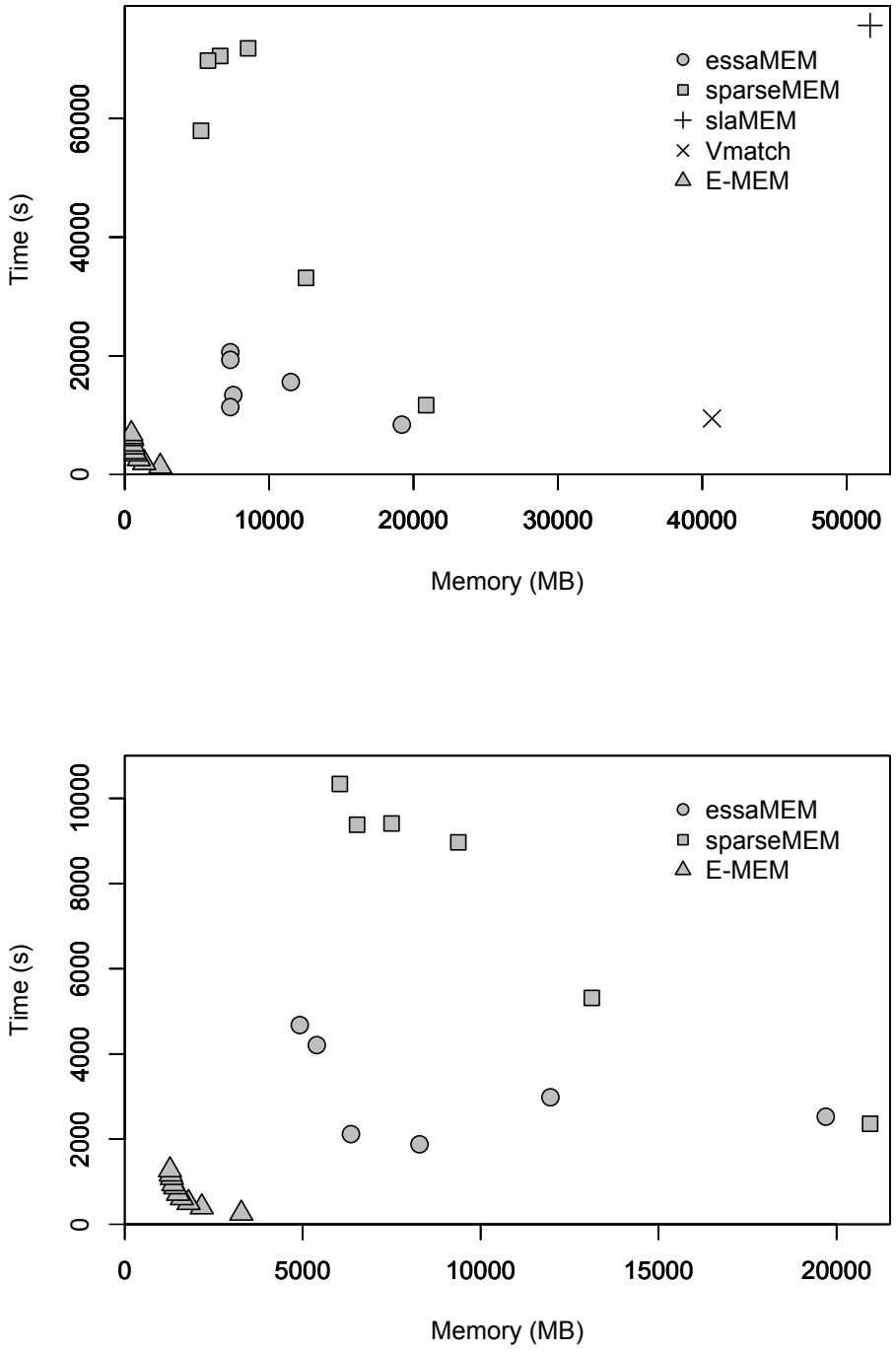

Figure 2.11: *Homo sapiens* vs *Pan troglodytes*; MEMs of minimum length 300. The top plot is for serial mode, the bottom for parallel. Note the different scale of the plots.

2.12.4 Triticum aestivum vs Triticum durum

For the last test, two wheat genomes were tested that are larger than the mammalian genomes used in the previous tests. Only E-MEM and Vmatch could run this test and the results for minimum MEM length 100 and 300 are presented in Tables A.5 and A.6 respectively. Other programs either reported wrong number of MEMs or crashed during the execution.

Minimum MEM length 100

For minimum MEM length 100 - 3,668,632 MEMs are reported by all programs. E-MEM outperforms Vmatch with a great margin. Figure 2.12 shows a comparison for time and memory of each tool. Note that this figure is slightly different than previous figures for similar plots. In Figure 2.12, both serial and parallel results are presented in the same plot for E-MEM, since Vmatch program cannot be run in parallel mode. E-MEM runs this test very efficiently and beats Vmatch by a big margin. All E-MEM results are close to origin, while Vmatch result is on top right corner. In serial mode, it uses less than 1GB of memory and in parallel it requires only 16 min and 2GB of memory. Refer Appendix A for detailed results.

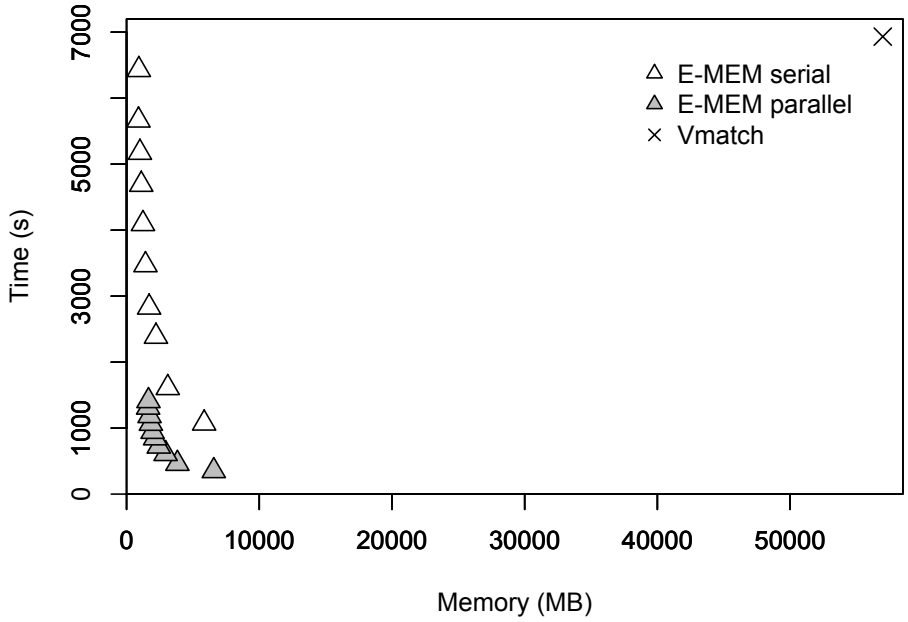

Figure 2.12: *Triticum aestivum* vs *Triticum durum*; MEMs of minimum length 100.

Minimum MEM length 300

For minimum MEM length 300- 613,607 MEMs are reported. Similar to Figure 2.12, results for serial and parallel mode for E-MEM are combined with Vmatch serial results in Figure 2.13. Again, E-MEM outperforms Vmatch in every aspect of performance. Refer Appendix A for detailed results.

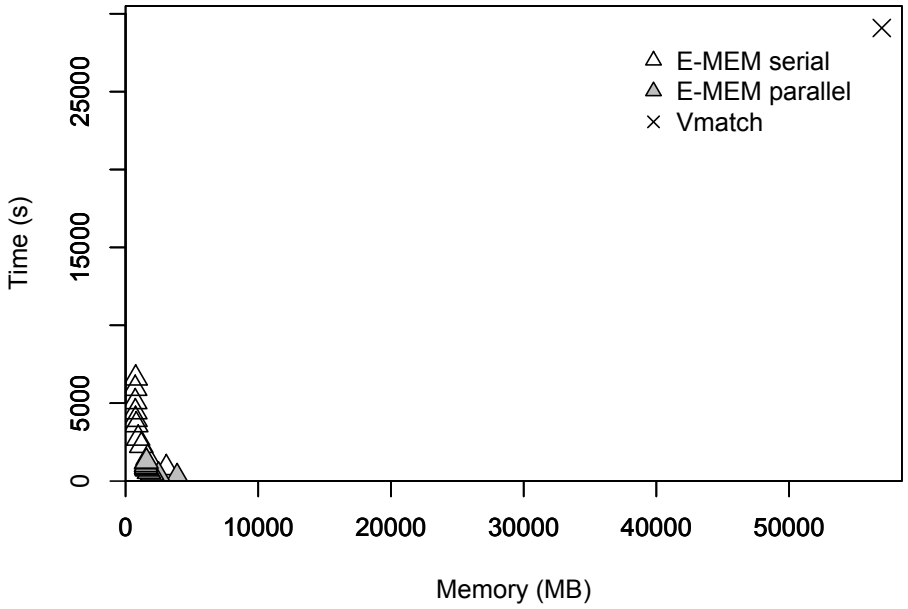

Figure 2.13: *Triticum aestivum* vs *Triticum durum*; MEMs of minimum length 300.

2.13 Conclusions

E-MEM provides an efficient solution for finding MEMs between arbitrarily large genomes. It can be used as a stand alone program or as a drop-in replacement for the MUMmer3 software package [34]. The test results show that E-MEM is many times faster than other state-of-the-art programs and also requires significantly less memory. The split parameter capability of E-MEM makes it unique - which is very useful when computing MEMs for very large genomes. Results show that memory reduction techniques used in other programs are not as effective and most programs fail to run genomes beyond 4 Gbp.

Chapter 3

Assembly Evaluation: LASER

3.1 Background

Over the last decade, N50 metric has been used to evaluate the quality of genome assembly. N50 is defined as the length of the shortest contig for which longer and equal length contigs of at least that length cover at least 50% of the assembly. A related and widely used parameter is NG50, which is similar, except that it uses genome size instead of assembly size. Figure 3.1 shows a dummy example of N50 computation. The contigs are arranged in decreasing order of size. The contig sizes are added in the order shown until it accounts for 50% of the total assembly size. The last or the smallest contig in this set is the N50 value of the genome assembly.

Recently, it was realized that programs tend to produce longer contigs either erroneously or purposely, giving an impression of better assembly in terms of N50 and NG50 matrices. For accurate evaluation of genome assemblies, the generated contigs are aligned to a reference genome and only the aligned blocks of contigs are used to compute N50 and NG50 values, which are called NA50 and NGA50 respectively. The alignment with a reference genome also provides opportunities to further classify misassemblies. The misassemblies can be classified as misjoins, indels and mismatches. Misjoins are the most undesirable type of misassembly

Figure 3.1: N50 example.

in which two far apart fragments are joined producing longer contigs. The three types of misjoins are *inversion*, *relocation* and *translocation*. An inversion occurs when the orientation of a contig is inverted with respect to the reference, a relocation occurs when a contig is misplaced within a single chromosome and a translocation occurs when a contig is misplaced into different chromosomes.

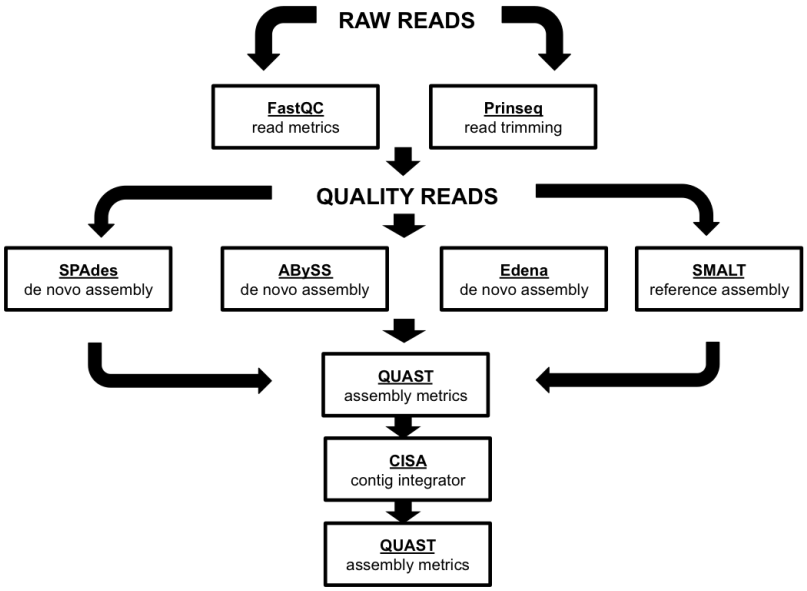

Figure 3.2: QUAST genome assembly evaluation flow [24].

Figure 3.2 shows a typical genome assembly process with assembly evaluation stages. The assembly evaluation is performed on contigs produced by genome assembler and scaffolds

produced by scaffolding programs such as Minimus [56] or Bambus [47]. Scaffolding is a process in genome assembly in which contigs are arranged in right order and orientation. The contigs will either overlap or separated by gaps of known length. The most popular genome assembly evaluation programs are QUAST [25] and GAGE [50]. However, all these programs are very slow for evaluation of large genomes. It is important to perform evaluation efficiently along with its accuracy, therefore there is a need to develop newer algorithms.

The de novo genome assembly problem is one of the most fundamental problems in Bioinformatics. Both, NGS and third generation sequencing technologies have limitations which pose challenges in assembling genomes. The NGS technologies produce very short reads and have non uniform coverage. Similarly, third generation sequencing technology has very high error rate. Various assembly algorithms using NGS, third generation or hybrid sequencing data have been designed and they all claim to produce longer and more reliable contigs. Methods to reliably assess the quality and accuracy of these assemblies are required for choosing the most suitable assembly for downstream analysis. Many approaches with new evaluation metrics have been proposed in recent years. In the following section, we discuss the features and metrics offered by QUAST evaluation program.

3.2 QUAST Introduction

The QUAST [25] (*QUality ASsessment Tool*) program is a state-of-the-art genome and meta-genome assembly evaluator which has introduced many new metrics. *Meta-genome assembly* is a fairly new research field and it is focused on the analysis of sequencing data derived from mixtures of organisms. QUAST can evaluate assemblies both with and without a reference genome. It produces many reports and plots to better understand the quality. The biggest drawback of QUAST was very high time and memory requirement which was a bottleneck for many large genome assembly evaluations. We tested QUAST performance on various human genome assemblies and it required around 4 days and 120GB of RAM to finish the evaluation.

This is a problem, since the actual assembly generation takes much less time to produce.

LASER (*Large genome ASsembly EvaluatoR*) [29] is a new genome assembly evaluator based on QUAST, but it is much faster than QUAST and requires half the memory. Traditionally, assembly evaluation programs have been heavily relying on Nx ($0 \leq x \leq 100$) statistics, which can be artificially increased by concatenating contigs, which can happen either erroneously or deliberately. The Nx approaches will fail to capture this error giving an impression of better assembly in such cases. QUAST produces an evaluation statistics called NAx, which computes Nx after aligning contigs with a reference genome. The NAx is computed using only the aligned blocks of contigs. Any contigs which are joined artificially, are broken at a point of misassembly. Some of the commonly used metrics for assembly evaluation are summarized in the following subsections.

3.2.1 Contig sizes

The metrics in this category can be evaluated with or without a reference genome. The program allows ignoring very small contigs which are not useful.

- *No. of contigs:* Total number of contigs in assembly
- *Largest contig:* The largest contig in the assembly
- *Total length:* Total length of all the generated contigs in base pairs
- *Nx ($0 \leq x \leq 100$):* The length of the largest contig L, such that adding all contigs of length $\geq L$ accounts for x% of total length
- *NGx:* The length of the largest contig L, such that adding all contigs of length $\geq L$ accounts for x% of reference genome length

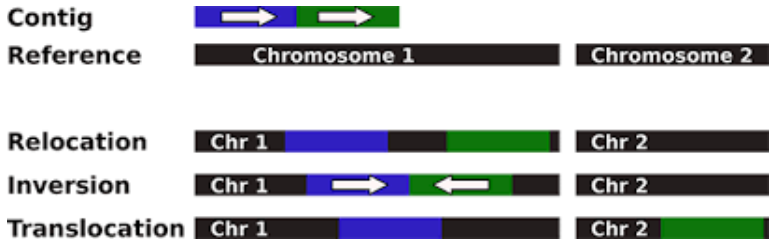

Figure 3.3: Structural Variations [25]

3.2.2 Misassemblies and structural variations

The metrics under this category are produced after aligning contigs with a reference genome. Any alignment differences are attributed to misassemblies, structural variations or due to sequencing errors. Structural variations can further be divided into *relocations, inversions* and *translocations*. A relocation is a breakpoint where the left flanking sequence aligns some distance away from the right flanking sequence. An inversion is a breakpoint where the flanking sequence aligns with its reverse complement. A translocation is a breakpoint where flanking sequences are some distance away and align on a different chromosome. Figure 3.3 shows a pictorial representation of structural variations.

- *No. of misassembled contigs:* The number of contigs which are joined erroneously and are broken during the alignment
- *Misassembled contig length:* The total number of bases in contigs with misassemblies
- *No. of unaligned contigs:* Number of contigs which could not be aligned with the reference

3.2.3 Genome representation

The metrics in this category provide an insight into the genome. These metrics cannot be produced without a reference genome.

- *Genome fraction (%):* The percentage of total aligned bases with respect to genome size.

- *Duplication ratio:* This is the ratio of total number of aligned bases in assembly with total number of aligned bases in reference. The ratio > 1 indicates that repeat count has been over estimated.

- *GC (%):* The GC content of an assembly is the percent of nucleotides which account for a base G or C. This metric can be computed without a reference genome.

- *No. of mismatches per 100 kb:* The average number of mismatches per 100 kb.

- *No. of genes and operons:* The number of genes and operons are predicted based on user provided annotations. Operons is group of genes or a segment of DNA that functions as a single transcription unit.

3.2.4 NAx and NGAx

These are new powerful metrics which are similar to Nx and NGx statistics where x varies from 0 to 100. The idea is to split original contigs at a breakpoint identified in previous definitions. A breakpoint can occur due to unaligned blocks in the contig or due to structural variations. Once the aligned contig blocks have been identified, the NAx and $NGAx$ are computed as the Nx and NGx statistics using these contigs.

3.2.5 Visualizations

Here, we are also including some of the plots supported by QUAST. These plots have been generated for dataset H_5 in Table 3.1. These are dynamic plots and when viewed in the browser, the details appear by hovering the mouse along the plot.

Cumulative length

This plot shows the growth of total contig length when arranged in decreasing order of size.

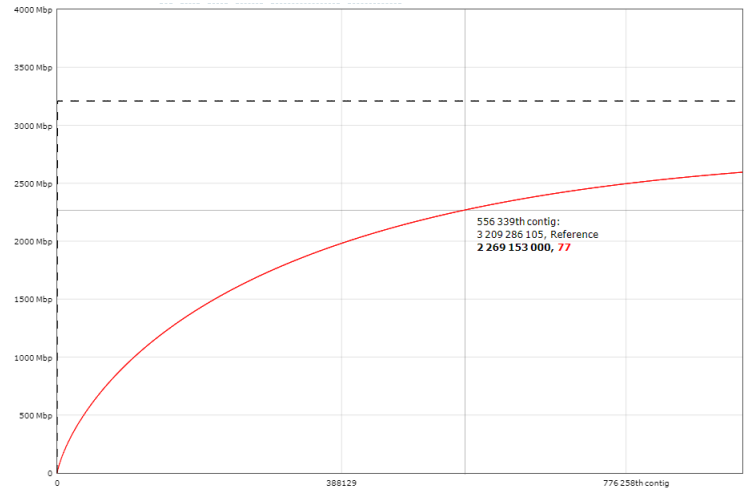

Figure 3.4: Cumulative contig length

Nx plot

This plot shows Nx values where x varies from 0 to 100%.

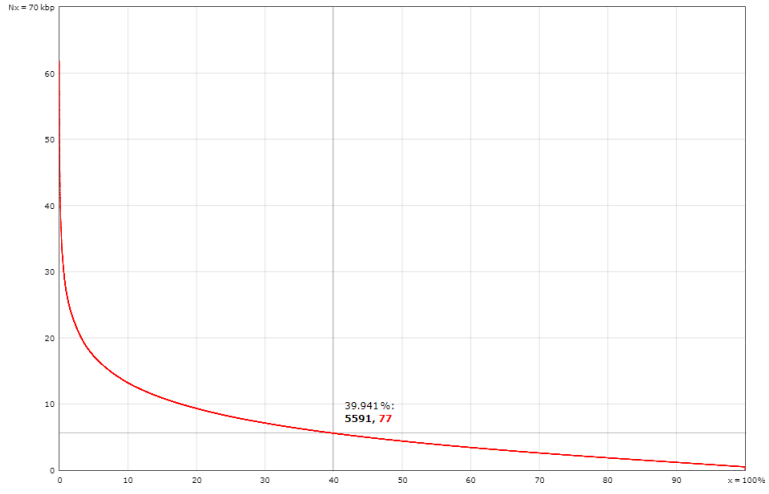

Figure 3.5: *Nx* values

NAx plot

This plot shows *NAx* values where *x* varies from 0 to 100%.

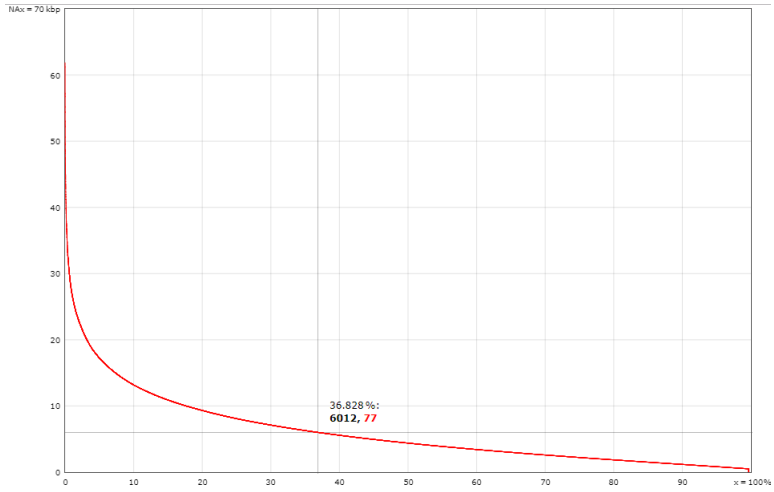

Figure 3.6: *NAx* values

NGx plot

This plot shows *NGx* values where *x* varies from 0 to 100%.

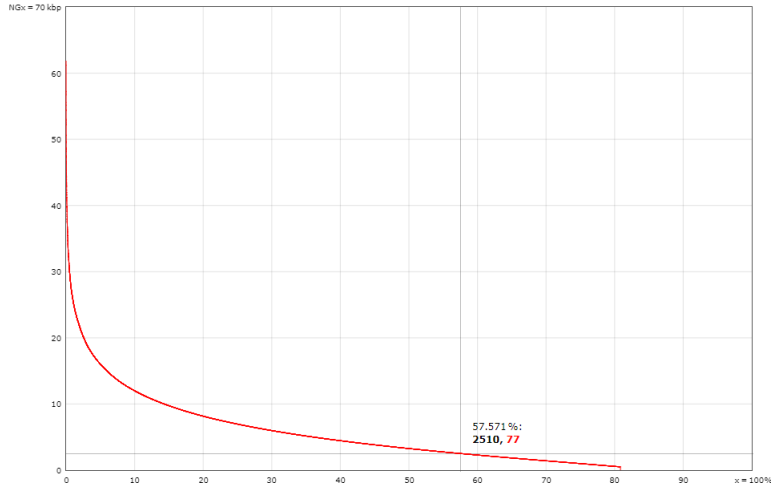

Figure 3.7: *NGx* values

NGAx plot

This plot shows *NGAx* values where *x* varies from 0 to 100%.

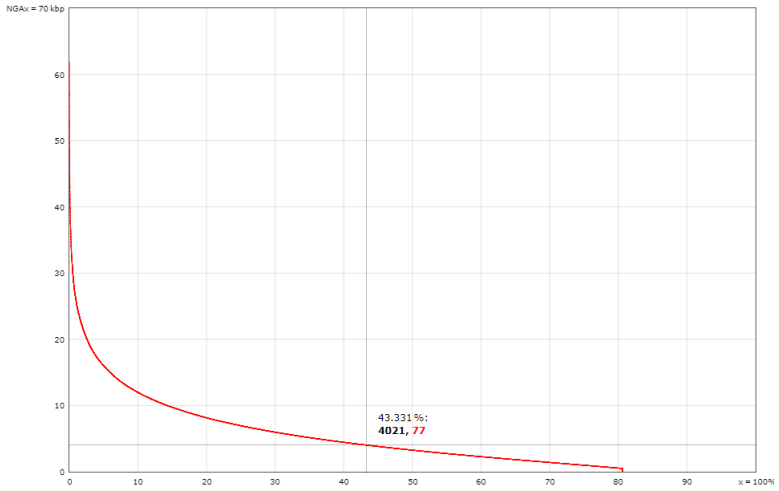

Figure 3.8: *NGAx* values

GC content plot

This plot shows the distribution of GC content in certain range. The x-axis has the GC content percent value while y-axis has the number of contigs for corresponding GC percent value. This value on y-axis is computed by finding GC content of non-overlapping window of size 100. The GC content plot shows two curves when reference genome is available. The curve in red indicates the GC content plot for the assembly while the dotted line curve is for the reference genome. For meta-genome assemblies, the plot will show multiple peaks, one corresponding to each genome.

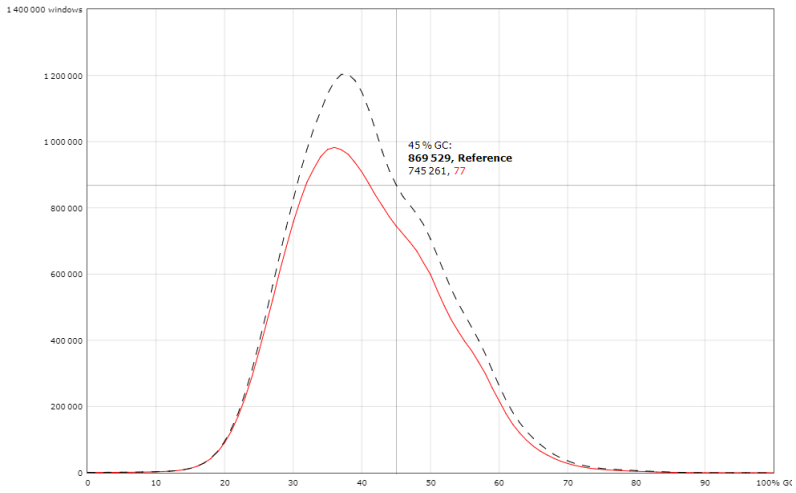

Figure 3.9: *GC* content

3.3 LASER Improvements

QUAST is the current state-of-the-art in assembly evaluation program which provides a thorough evaluation of assemblies by means of many new metrics and visualizations as discussed in Section 3.2. However, for large genomes, the high time and memory usage requirements are still a bottleneck for many researchers. We tested five different data sets of human genomes and in most cases, it requires over 4 days and 120 GB of RAM to assess the quality of a single human genome assembly.

LASER has been designed and implemented to inherit the advantages of QUAST while improving its performance. The following subsections describe the major improvements implemented in LASER and compare its performance with QUAST on several human datasets.

3.3.1 E-MEM integration

One of the key changes in LASER is the use of E-MEM [30], in place of NUCMmer [34]. NUCMmer is one of the most widely used program for MEM computation and genome alignment. NUCMmer uses suffix tree data structure for creating an index which requires huge amount of memory and is very slow. E-MEM, as discussed in Chapter 2, is currently the

best program for MEM computation and it is many times faster than NUCMmer, which gives LASER a significant improvement in memory and run time.

3.3.2 Code remodeling

During the E-MEM integration for LASER, it was realized that there are many data structures in QUAST which can be enhanced to further improve the performance and reduce the memory requirements. The QUAST code is written in Python which is an interpreted language and could be slow if not used properly. Efficient data structures and functions were used to remodel the QUAST code. The major changes include simplifying nested dictionaries and replacement of class objects with simple tuple based data structures. The changes improved performance and drastically reduced the memory requirements.

3.3.3 NUCmer changes

The genome evaluation process requires the alignment of contigs with reference genome. The alignment information is used for identification of *Single Nucleotide Polymorphisms* (SNPs) and *indels*. This is achieved by another utility from NUCMer tool-set called *show-snps*. The analysis of the utility with GNU profiler revealed functions which were consuming excessive amount of time. Minor changes including some code rearrangements were done to improve the performance of *show-snps* utility. The NUCmer was also modified to use temporary files instead of pipes.

3.4 Results

The performance of LASER and QUAST was compared on five different variants of human genome sequencing data. The data set details are provided in Table 3.1. Performance evaluation has been performed using the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca) and Compute Canada. All tests were

performed on a DELL PowerEdge R620 computer with 12 cores Intel Xeon at 2.0GHz and 256GB of RAM, running Linux Red Hat, CentOS 6.3.

Table 3.1: Sequencing data used for comparison

Datasets	Organism	Accession number	Read Length	Number of reads	Coverage
H_1	*Homo sapiens*	SRR1302280	101	1,287,175,558	41
H_2	*Homo sapiens*	ERR194146	101	1,626,361,156	51
H_3	*Homo sapiens*	ERR194147	101	1,574,530,218	50
H_4	*Homo sapiens*	ERR324433	101	1,614,713,636	51
H_5	*Homo sapiens*	ERX069505	101	1,708,169,546	54

Table 3.2: Assembly generation and evaluation time comparison

Datasets	*k*-mer size	Assembly time (s)	Evaluation time (s)
H_1	71	64,880	184,356
H_2	71	60,480	359,615
H_3	71	81,720	351,867
H_4	71	71,640	358,743
H_5	65	77,400	349,227

The tests were initially performed on small datasets and significant improvements were noticed. However, the idea of LASER was to improve performance on large datasets. First, assemblies were generated using the SOAPdenovo2 [36] assembler for all the datasets mentioned in Table 3.1. These assemblies were then evaluated using QUAST. The time comparison of assembly generation and evaluation are shown in Table 3.2. QUAST requires 4 days to evaluate an assembly which took less than a day for generation. This makes it nearly impossible to use QUAST for evaluating the quality of an assembler being developed.

Table 3.3 gives the time and memory comparison between QUAST and LASER on the SOAPdenovo2 [36] assemblies produced from the datasets in Table 3.1. A better visual comparison is shown in Figure 3.10. LASER is 5.6 times faster than QUAST while using half the memory. For dataset H_1, the performance results are different from other datasets due to the smaller coverage for this dataset.

Dataset	Time (s)		Memory (GB)	
	QUAST	LASER	QUAST	LASER
H_1	184,356	35,715	101.1	50.9
H_2	359,615	64,368	123.3	61.2
H_3	351,867	62,879	122.3	60.3
H_4	358,743	63,917	124.3	61.0
H_5	349,227	62,291	121.0	59.8

Table 3.3: QUAST and LASER comparison

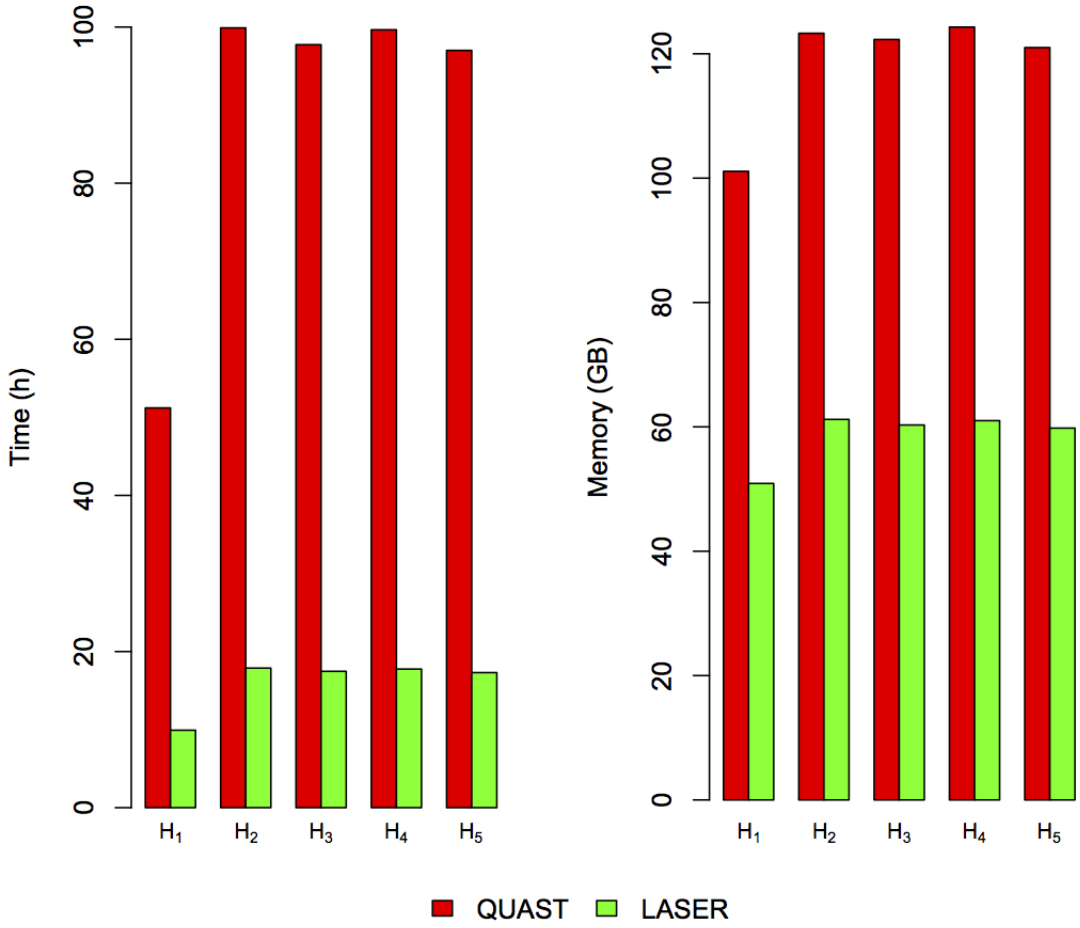

Figure 3.10: Visual performance comparison of QUAST and LASER

3.5 Conclusions

Genome assembly evaluation is an essential step in assessing the quality of an assembly. Unfortunately, it is often ignored or done improperly due to lack of understanding or absence of required computing resources. With the introduction of LASER, proper evaluation can be performed efficiently. A detailed overview of genome assembly evaluation process and metrics is provided. LASER is an important program for the research community that improves on drawbacks of QUAST and provides the benefit in efficient analysis of assemblies and its annotations. After the LASER publication, QUAST program has replaced MUMmer with E-MEM in order to improve performance. LASER has additional changes which still makes it faster than QUAST.

Chapter 4

Genome Alignment: HISEA

4.1 Background

The information in DNA is encoded in large sequences. To decode this information, techniques are needed to compare two or more sequences. *Sequence Alignment* is a method which is used to compare and understand the similarity or differences among sequences. The method involves arranging sequences such that similar regions are aligned to each other. These regions are separated by one or more gaps, whenever mismatches are encountered. Figure 4.1 shows an alignment between sequences TGGTTACT and TAGTAGTTACT. The gaps or mismatches in the alignment are shown with an underscore character.

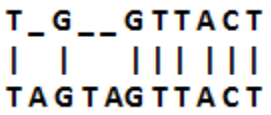

Figure 4.1: Alignment example.

A very small sequence can be aligned by hand, see example in Section 1.4. However, complex genomic sequences require computational methods to be developed for faster and optimal alignment. Computational methods for sequence alignment fall in two categories - *Global Alignment* and *Local Alignment*. Global Alignment is used to compare similar sized

sequences and it forces the alignment to span the entire length of the sequence. On the other hand, a local alignment is used to find the shared subsequences within two otherwise divergent sequences. Local alignments are preferred but it is often more difficult to compute than global alignments. Figure 4.2 shows a conceptual representation of two methods.

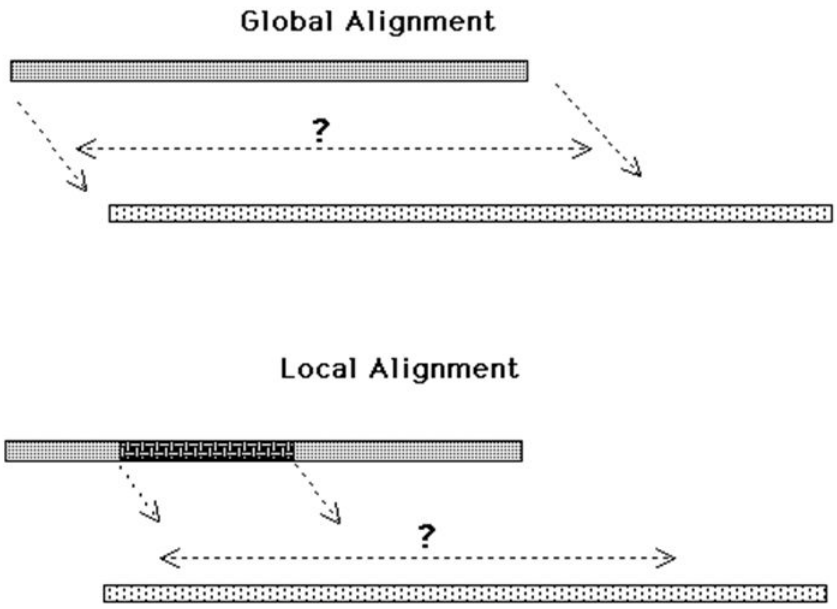

Figure 4.2: Global vs Local Alignment [14]

The first global alignment algorithm was developed by Saul B. Needleman and Christian D. Wunsch [45] in 1970, which was based on dynamic programming. In 1981, Temple F. Smith and Michael S. Waterman [55] proposed a local alignment algorithm based on a similar idea. The algorithms based on dynamic programming are guaranteed to find an optimal alignment. However, the algorithm has quadratic time complexity and does not scale up very well with increasing sequence sizes. Given two sequences for alignment, $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$ of length n and m respectively, the recurrence relation for Smith-Water algorithm is as follows:

$$T_{ij} = \max \begin{cases} T_{i-i,j-1} + s(a_i, b_j), & \text{where } s(a_i, b_j) = \begin{cases} +1, & \text{if } a_i = b_j \\ -1, & \text{if } a_i \neq b_j \end{cases} \\ T_{i-1,j} - gapPenalty, & \text{and } (1 \leq i \leq n \text{ and } 1 \leq j \leq m) \\ T_{i,j-1} - gapPenalty, & \\ 0 & \end{cases}$$

Here T is a scoring matrix of size $(n+1) * (m+1)$ and T_{ij} represents a cell in this matix at an intersection of row i and column j. The similarity score of the two elements is defined by $s(a_i, b_j)$ and $gapPenalty$ is the penalty for opening or extending the gaps. In general, matches are assigned positive scores, and mismatches are assigned negative or relatively lower scores. The recurrence shows an example of similarity scoring where matches get +1 and mismatches get -1. A gap is created when there is an insertion or deletion in one sequence with respect to other. In order to make accurate alignment decisions, gaps are penalized via various gap penalty scoring methods. A simplest gap penalty method is to assign a *constant* negative score to every gap, regardless of its length. This scheme is inaccurate because it does not account for the length of insertion and deletion. A *linear* gap penalty solves this problem by accounting for the length of the alignment. The most widely used gap penalty function is the *affine* gap penalty. The affine gap penalty combines the advantages of both constant and linear gap penalty into the scoring system. This scheme closely model the biological evolution process. Vingron *et al.* [58] provides a detailed discussion on gap penalty schemes and its implication on sequence alignment.

The algorithm start by constructing a scoring matrix T of size $(n+1)*(m+1)$ and initializes its first row and first column with 0s. Then, starting with top left corner, values for each T_{ij} are computed by using the recurrence relation. It is clear from the recurrence that smallest possible value for a cell is 0, meaning that sequences up to this position have no similarities. Table 4.1 shows an example of scoring matrix computed for alignment of DNA sequences TGGTTACT

and TAGTAGTTACT. The similarity score of +1 for a match and -1 for a mismatch is used. A linear gap penalty score of -1 is used for both gap opening and extension.

Table 4.1: Smith-Waterman alignment for sequences TGGTTACT and TAGTAGTTACT

	-	**T**	**G**	**G**	**T**	**T**	**A**	**C**	**T**
-	0	0	0	0	0	0	0	0	0
T	0	1	0	0	1	1	0	0	1
A	0	0	0	0	0	0	2	1	0
G	0	0	1	1	0	0	1	1	0
T	0	1	0	0	2	1	0	0	2
A	0	0	0	0	1	1	2	1	1
G	0	0	1	1	0	0	1	1	0
T	0	1	0	0	2	1	0	0	2
T	0	1	0	0	1	3	2	1	1
A	0	0	0	0	0	2	4	3	2
C	0	0	0	0	0	1	3	5	4
T	0	1	0	0	1	1	2	4	6

The optimal local alignment is found by tracing back from the highest score value to its source recursively until 0 is encountered. For example in Table 4.1, the highest score is 6 and traceback is shown with arrows starting at this position. Similarly, the second best local alignment is obtained by tracing back from second highest score excluding the best alignment.

Over the years, focus has shifted on heuristic algorithms for sequence alignment which are very fast, but do not guarantee an optimal alignment. Heuristic alignment algorithms have many applications and are widely used in Bioinformatics. One of the most popular heuristic alignment algorithm is *Basic Local Alignment Search Tool* (BLAST) [4]. BLAST was designed for NGS technologies. It does not work well with long reads due to high error rate. The long

read sequencing technologies have error rate of 10-15% and the most dominant errors are *indels*. This is different from NGS sequencing technologies where read lengths are short and error rate is below 2%. Hence, the alignment algorithms developed for NGS technologies cannot be directly utilized for long reads.

Recently, many new alignment algorithms have been proposed. Typically, all alignment algorithms for long reads follow the same methodology - find the candidate overlap, estimate overlap distance and identify the approximate overlap region. In the following sections I briefly discuss some of the recently published long read alignment algorithms.

4.1.1 BLASR

Basic Local Alignment with Successive Refinement (BLASR) [16] was the first algorithm used for long reads alignment. The algorithm was developed as a long read mapper and can be used for alignment with some parameter tuning. BLASR can use either a suffix array or FM-index to find the exact matches between sequences. Figure 4.3 shows an overview of BLASR alignment process. The exact matches (or *k*-mers) are shown in colored arrows.

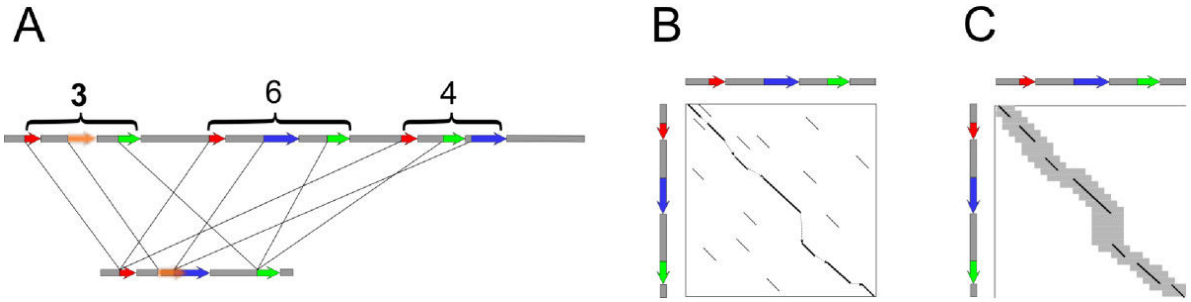

Figure 4.3: Overview of BLASR algorithm; Chaisson *et al.* [16]

All exact matches are clustered and a score is assigned to each cluster. The scores for three clusters are shown as 3, 6 and 4 in figure 4.3 (A). All clusters with a score above the minimum threshold are aligned using sparse dynamic programming as shown in figure 4.3 (B). High scoring alignments from the last step are realigned using dynamic programming over a subset of cells guided by sparse dynamic programming alignment - Figure 4.3 (C). BLASR

has high computing requirements as it uses dynamic programming which is known to have large runtime requirements. BLASR is used in the PBcR assembly pipeline which was the first genome assembly pipeline for long reads data [32].

4.1.2 DALIGNER

DALIGNER [44] was the first program designed for Pacbio read-vs-read alignment. To determine the similarity between two sequences, the algorithm finds a set of *k*-mers in each read, sorts each set according to *k*-mers and merges common *k*-mers to a new set. These *k*-mers are later extended to find local alignment using an algorithm of progressive "waves" of furthest reaching points [43].

DALIGNER has many techniques implemented for efficiency. It utilizes 3-level cache hierarchy architecture of modern computers to achieve faster memory fetch operations. The radix sort algorithm is used to increase the sorting performance and thread level parallelization is implemented for runtime efficiency. DALIGNER is used in the FALCON [10] assembly pipeline.

4.1.3 GraphMap

GraphMap [57] was originally designed as a read mapper for *Oxford Nanopore Technology* (ONT). It enables an overlap detection mode under an option '-owler'. GraphMap is the only program which uses *spaced seed* for detecting similarity between sequences. The concept of space seed was introduced in PatternHunter [39], where the seeds do not consist of consecutive matches. The seed used in PatternHunter is 111*1**1*1**11*111, which looks for 18 consecutive nucleotides in each sequence such that only nucleotides at 1's position are required to match. GrpahMap uses a hard coded spaced seed 111111*111111 of size 13. The space seed matches 13 base pairs and allows a mismatch or an indel at the position number 7. The identified *k*-mers are filtered to find a longest common subsequence of these matches. The bounds of the longest subsequence is returned as the alignment bound.

4.1.4 MHAP

MinHash Alignment Process (MHAP) [6] implements MinHash algorithm by Broder [13] to detect similarity between two sequences. MinHash is an efficient hashing technique which uses multiple hash functions and stores the smallest hashed values in a *sketch* list. Thus, each sequence is represented by its own *sketch* irrespective of the read length. The *sketch* size depends on the number of hash functions used. Higher number of hash functions increases sensitivity but has a negative impact on the performance. An example of minHash is shown in Figure 4.4. The example uses four hash functions for two sequences S_1 and S_2. The k-mer size is 3. The *sketch* for sequences are shown in Figure 4.4 (c).

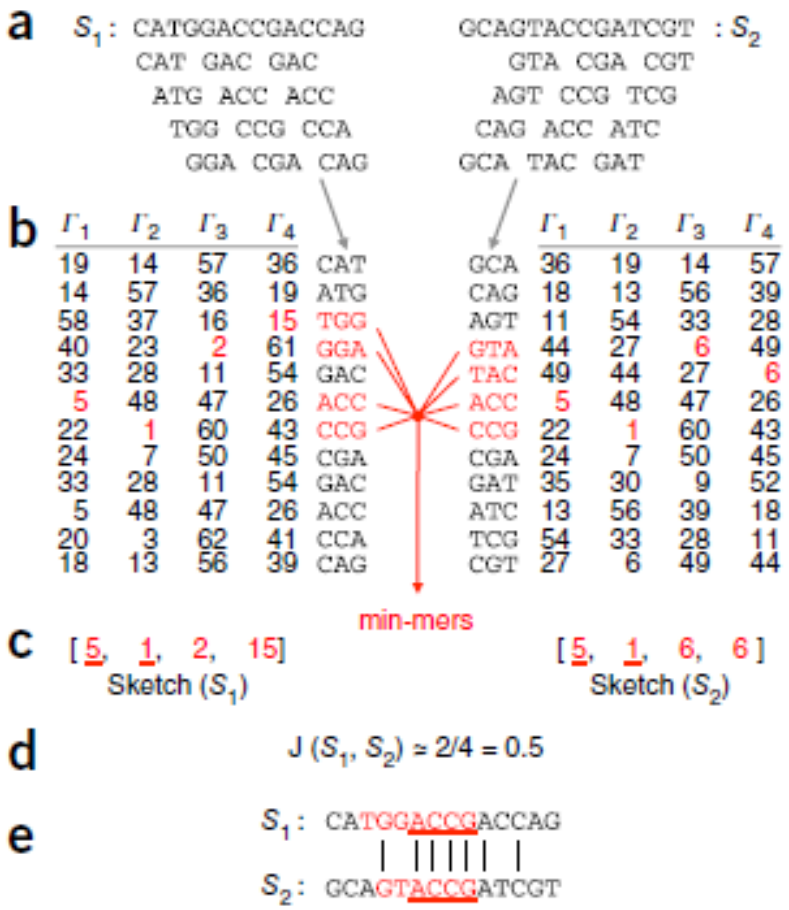

Figure 4.4: MinHash overview; Berlin *et al.* [6]

Similarity is computed by the Jaccard index over the sequence of *sketches*. The *Jaccard index* [26] is defined as the size of the intersection divided by the size of the union of the sample sets. Given two sets A and B, the Jaccard index $J(A, B) = |A \cap B| \div |A \cup B|$ and $0 \leq J(A, B) \leq 1$. The overlapping region is identified by computing the median of relative positions of matches. These are further validated against *sketches* of smaller k-mer size for correctness. In practice, MHAP is very fast due to the use of MinHash, but requires more memory as the code has been implemented in JAVA. MHAP is used in Canu [33] assembly pipeline.

4.1.5 Minimap

Minimap [35] is the most recent aligner and it improves on many ideas introduced in previous aligners. It uses hash table to store *k*-mers similar to the alignment programs BLAT [27]. It uses minimizers, Roberts et al. [48], an idea similar to *sketch* in MHAP for reduced representation of sequences. A *minimizer* is the smallest *k*-mer in a window of *w* consecutive *k*-mers. Similar to DALIGNER, it uses sorting to reduce heap allocation and avoid cache misses. The core algorithm flow is similar to other algorithms where minimizers are used to identify the similarity between sequences. A longest increasing subsequence of *k*-mers gives the bounds of the approximate alignment. Minimap is used in the Miniasm assembly pipeline [35].

4.2 HISEA Introduction

An essential step in assembling SMRT data is the detection of alignments, or overlaps, between reads. High error rate and very long reads in the Pacbio sequencing data make this a much more difficult problem as compared to Illumina data. We present a new pairwise read aligner, or *overlapper*, HISEA (HIerarchical SEed Aligner) [31] for SMRT sequencing data. The HISEA algorithm has the best alignment detection sensitivity among all programs for SMRT data. We compared the sensitivity of our aligner with BLASR [16], DALIGNER [44], GraphMap [57], MHAP [6], and MiniMap [35]. Note that the terms "alignment" and "overlap" are used interchangeably.

The comparatively high cost of SMRT sequencing has prevented its widespread use. It is very expensive to sequence large genomes with high coverage using SMRT technology, therefore it is still beyond the reach of many research labs. Recently, Koren *et al.* [33] showed that their Canu assembler can generate assemblies using only 20x coverage that are comparable with 150x coverage hybrid assemblies generated with SPAdes [5]. They have also shown that it can achieve maximum assembly continuity around 50x coverage. As indicated by Koren *et al.* [33], Canu is currently the best pipeline. Therefore, we have incorporated HISEA

in this assembly pipeline, replacing the MHAP aligner [6] with HISEA. We have compared the two pipelines, Canu+MHAP and Canu+HISEA for five organisms, *E.coli*, *S.cerevisiae*, *C.elegans*, *A.thaliana*, and *D.melanogaster* at two coverage levels: 30x and 50x. The pipeline using HISEA is shown to produce better assemblies for both coverage levels. Moreover, the Canu+HISEA assemblies for 30x coverage are comparable with those of Canu+MHAP for 50x coverage.

HISEA software is implemented in C++ and OpenMP. It can be used as a stand alone aligner or as an all-vs-all read aligner in other assembly pipelines.

4.3 HISEA algorithm

Let $\Sigma = \{A, C, G, T\}$ be the DNA alphabet; Σ^* is the set of all DNA sequences, that is, all finite strings over Σ. Assume two sets of reads: the set of reference reads, $R = \{r_1, r_2, ..., r_n\} \subset \Sigma^*$, and the set of query reads, $Q = \{q_1, q_2, ..., q_m\} \subset \Sigma^*$. A k-mer is a string of length k over Σ.

4.3.1 Storing reads and hashing the reference set

All reads and their reverse complement are stored in memory. In order to keep memory requirements low, each read r_i is encoded using 2 bits per nucleotide and stored as an array of unsigned 64-bit integers, that is, as blocks of 32 nucleotides. This reduces memory requirement by 4 folds compared to storing them as one byte per character. The reverse complement of r is stored in the same array and it starts at the next unsigned 64-bit integer. A precomputed 16-bit reverse complement array of all possible values is used to quickly compute the reverse complement of reads. Since the size of the sequence is known, accessing the reverse complement simply requires movement of a pointer to the right location.

All k-mers that occur in reads of R and its reverse complement are quickly computed using bitwise operations and bit masking. This is achieved by sliding a window of k-mer size through the sequence and extracting k-mers using bit operation. These k-mers are stored in a hash table

using double hashing technique, similar to what we discussed in Section 2.5. The hashing is implemented in parallel using OpenMP directives. As a first step, a temporary hash is used for storing the k-mers by dynamically growing the hash table. Once all the k-mers are added, the final hash table is created by moving k-mers in this table. Each hash table entry stores the value of the k-mer and a pointer to a second hash table. The second hash table is a dynamic hash table (a map container class from *Standard Template Library*), which stores the set of read ids r_j, and positions within r_j, where this k-mer occurs. As a pre-filtering step, any k-mers appearing more than a specified upper bound threshold or less than a lower bound threshold are ignored for hashing. Ignoring these k-mers does not impact the alignment results. Instead, it helps in reducing memory and improving performance in later stages of the algorithm. The default value of the lower bound and the upper bound is 2 and 10,000 respectively.

4.3.2 Searching the query set

The k-mers occurring in the query read set Q are not stored; they are quickly computed as needed using bit operations. Similar to reference storing, this is achieved by using a window of k-mer size and going over the entire sequence. Again, the use of bit operations makes the computation very efficient. Next, each k-mer from the query set is efficiently searched in the hash table built for the set R. Every time a matching k-mer is found in the hash table, the corresponding reference read identifier and its position are recorded. The information is stored in a complex data structure which consists of a hash containing array of vectors. All the basic data structures are directly used from STL container classes except the hashing as discussed in the previous section.

It is important to note that the reads in the query set are only searched in forward direction, as the reference hash contains k-mers in both forward and reverse directions.

4.3.3 Filtering and clustering

The next step in the process is to filter and cluster the k-mer matches. In the previous step, for each query $q \in Q$ and a reference read $r \in R$, the reference read direction and all matching k-mer positions are stored. For a pair of reads (q, r), further processing is considered only either in forward or reverse direction of r. The decision is taken based on the read direction of r which has a higher number of matching k-mers.

At this point, we have all k-mer matches of a given size between query sequence q and reference sequence r. Some of these matches appear just by chance or may not correspond to the best possible alignment. A clustering step is performed to group the k-mers such that each cluster contains a consistent set of k-mers. A *consistent* set of k-mer matches is defined as a set of all k-mer matches arranged in ascending order of their positions and are equidistant from neighboring k-mer matches within a defined threshold. The threshold is dependent on error rate which can also be controlled by a command line parameter. Figure 4.5 shows an example of all k-mer matches between read q and read r before and after clustering. The example shown here is one simple case; in reality many complex cases are possible where clustering is essential.

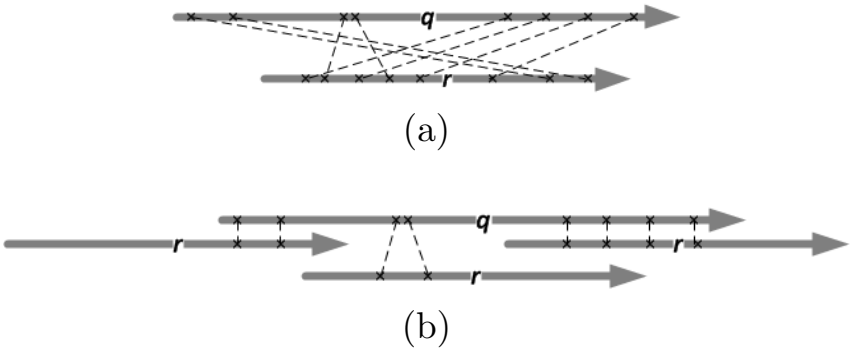

Figure 4.5: All k-mer matches between reads q and r before (a) and after (b) clustering.

Clustering is an essential step in identifying the best alignment out of multiple possible alignments. We report only the best alignment out of all possible alignments between a pair of reads. The initial matches can have contradictory information, such as the ones in Figure 4.5(a). The clustering phase involves collecting together consistent matches which helps in

determining the best possible alignment. Figure 4.5(b) shows the set of k-mers as divided into three consistent groups. It can be seen from the diagram that the rightmost cluster of k-mers is expected to produce the best alignment results.

Algorithm 4.3.1 gives the details of the clustering process. The input to the algorithm is an array V which contains all k-mer matches for a pair of reads (q, r). The input k-mer matches in V are sorted beforehand, first by query read positions and then by reference read positions. If the clustering algorithm fails to produce any meaningful clusters, we reverse the sort order i.e. first sort by reference read positions and then by query read positions and retry the algorithm. The algorithm uses two global parameters, *kmerSize* and *maxShift*. The parameter *kmerSize* is the size of the k-mers used for the initial hashing. The parameter *maxShift* is a user configurable parameter that accommodates the *indel* errors during k-mer matching, clustering and extension algorithms. The default value of this parameter has been experimentally determined to be 0.2 (or 20%). The output of the clustering algorithm is a set of matches, *ClusterArray*, segregated in groups such that each group has a consistent set of k-mers. Note that the first two values in *ClusterArray* store the left and right k-mer positions in V for that cluster. The third and fourth values are the number of matching base pairs and k-mer hit counts respectively.

Algorithm 4.3.1: CLUSTERKMERS(V)

```
global kmerSize, maxShift
local k ← 0, j ← 0, ClusterArray ← (0, 0, kmerSize, 1)
local found ← false , refDiff ← 0, queryDiff ← 0
for k ← 1 to V.size
do { found ← false
     for j ← 0 to ClusterArray.size
     do { refDiff ← (V[k].r − V[ClusterArray[j][1]].r)
          if (refDiff < 0)
            then continue
          queryDiff ← (V[k].q − V[ClusterArray[j][0]].q)
          if (queryDiff < 0)
            then continue
          if (refDiff and queryDiff within maxShift limits)
            then { found ← true
                   Update values in ClusterArray[j] }
     if (found = false )
       then { comment: Add new cluster in ClusterArray
              ClusterArray[j + 1] ← (k, k, kmerSize, 1) }
return (ClusterArray)
```

From the output of Algorithm 4.3.1, the cluster with the maximum number of matching base pairs is selected for further processing. The expected number of k-mer matches is estimated with the help of k-mer bounds in read q and read r; see Figure 4.6.

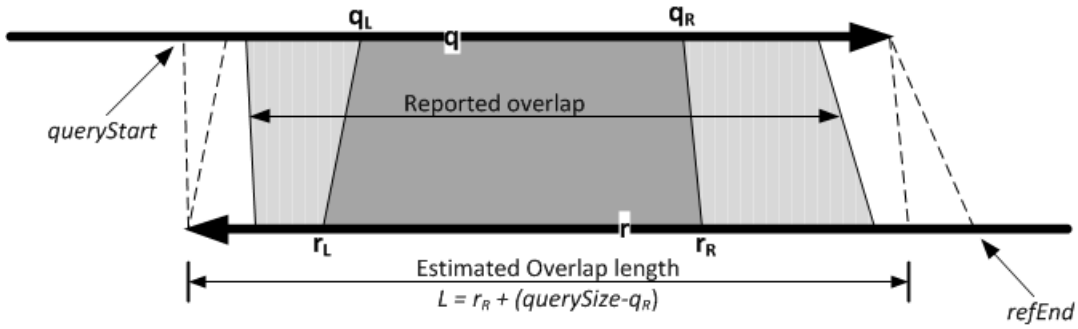

Figure 4.6: Computing the alignment. The dark grey region contains all k-mer matches and is extended by the light grey ones using k'-mer matches.

The leftmost and rightmost query k-mers start and end at positions q_L and q_R respectively. Similarly, the corresponding positions in the reference read are r_L and r_R. The alignment length is $L = r_R + querySize - q_R$, if there exists a perfect overlap between r and q. The number of k-mer hits in the overlapping region is approximated as a binomial distribution with probability $p = (1-e)^{2k}$ and L trials. Overlaps that have fewer k-mer matches than three standard deviations below the mean, that is, less than $\mu - 3\sigma = Lp - 3\sqrt{Lp(1-p)}$, are eliminated as having too low similarity. This procedure is employed several times during different steps of the algorithm and will be referred to as the $\mu - 3\sigma$ *criterion*.

4.3.4 Computing and extending alignments

The alignment between the two given reads starts at the boundary defined by the set of k-mers in the cluster identified by Algorithm 4.3.1, shown in dark grey in Figure 4.6. This region is extended using a smaller seed, that is, using k'-mer matches, for some $k' < k$. The default values are $k = 16$ and $k' = 12$. These values have been determined experimentally to produce reasonably good results for most datasets. HISEA implementation allows these values to be modified from the command line.

The first step is to compute the maximum bounds of the alignment considering the maximum amount of allowable indels in the overlapping region. This is given by the user configurable parameter *maxShift* mentioned above. As an example, for the situation depicted in

Figure 4.6, we set the maximum bounds for read q and read r as $(queryStart, querySize)$ and $(0, refEnd)$ respectively (see Figure 4.6) where:

$$\begin{aligned} queryStart &= q_L - (1 + maxShift)r_L \\ refEnd &= r_R + (1 + maxShift)(querySize - q_R) \end{aligned}$$

Algorithm 4.3.2: EXTENDALIGNMENT($queryBound, refBound, V$)

---

local $refDiff \leftarrow 0, queryDiff \leftarrow 0, hits \leftarrow 0, currIndex \leftarrow -1$

for $i \leftarrow 1$ **to** $V.size$

do
- **if** ($currIndex = -1$)
  - **then**
    - $refDiff \leftarrow |refBound - V[i].r|$
    - $queryDiff \leftarrow |queryBound - V[i].q|$
  - **else**
    - $refDiff \leftarrow |V[currIndex].r - V[i].r|$
    - $queryDiff \leftarrow |V[currIndex].q - V[i].q|$
- **if** ($refDiff$ and $queryDiff$ within $maxShift$ limits)
  - **then**
    - $estimate \leftarrow \mu - 3\sigma$
    - **if** ($hits \geq estimate$)
      - **then**
        - $hits \leftarrow hits + 1$
        - $currIndex \leftarrow i$
      - **else if** ($currIndex \neq -1$)**break**

if ($currIndex \neq -1$)

then return ($V[currIndex].r, V[currIndex].q$)

else
- **comment:** Could not extend bounds
- **return** (0)

At this point, we compute all k'-mer matches ($k' < k$) within three regions - two light grey

regions and one dark grey region as shown in Figure 4.6 using similar methods as discussed in Section 4.3.2 and 4.3.3. However, now it does not go through the clustering step as the region is already identified as the best alignment region based on k-mer matches. The k'-mers are computed to accurately estimate the extension of the alignment using the $\mu - 3\sigma$ criterion. Next, the dark grey region is extended by the light grey region on both sides as shown in Figure 4.6. Each new k'-mer match is added with the ones already found as long as they satisfy the $\mu - 3\sigma$ criterion. Algorithm 4.3.2 provides details of this process. The input bounds are either (q_L, r_L) or (q_R, r_R).

Finally, all the k'-mers within the initial region – dark grey colour in Figure 4.6 – are computed. Note also that the process is now guided by the original k-mers and therefore the clustering step is not required. The $\mu - 3\sigma$ criterion is applied once more to the total number of k'-mer matches for the entire overlap (light and dark grey). If the criterion is satisfied, the reads are considered to be overlapping and the alignment is reported.

Note that HISEA computes only the alignment boundaries, not the actual alignments. This does not guarantee the optimal alignment bounds but we show in Section 4.4 that HISEA algorithm produces alignments which are very close to optimal. We have compared HISEA algorithm with other programs, such as MHAP [6], Minimap [35] and GraphMap [57], all of which produce approximate alignment bounds. As we know, optimal alignment is computed using Smith-Waterman [55] dynamic programming algorithm and using it for all-vs-all read alignment will be very time consuming. Further, the heuristic approach provides a very efficient solution and the approximate alignment bounds reported are very close to optimal. It is important to understand that HISEA - generated alignments are sufficient for downstream usage, particularly in genome assembly algorithm using large read sequencing technologies.

4.4 Alignment evaluation method

The *EstimateROC* utility estimates the sensitivity, specificity and precision for the alignments reported. The original *EstimateROC* utility of Berlin *et al.* [6] relies heavily on BLASR mappings for the verification of reported alignments. This is not an accurate procedure since BLASR can make errors. Ideally, each alignment needs to be verified against the optimally computed alignment using the Smith-Waterman [55] dynamic programming algorithm. Since each program reports millions of alignments, verifying each and every alignment is very time consuming and hence not feasible. We used BLASR mapping to randomly choose a pair of reads where alignment exists and then evaluate it against the optimal alignment computed using Smith-Waterman dynamic programming algorithm. The *EstimateROC* utility provides a skeleton for estimating *sensitivity, specificity* and *precision*. We have modified or re-written all the functions provided by *EstimateROC* to suit our evaluation process . Figure 4.7 shows a relationship between true and false alignments.

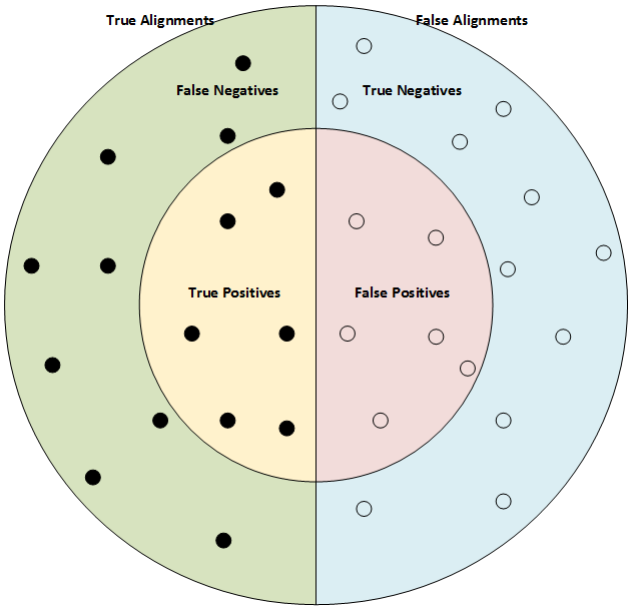

Figure 4.7: Relationship between alignments reported by program and real alignments

The smaller circle contains all the alignments reported by the program and it divides all alignments into four regions shown in different colors. An alignment in each region represents

one of the values corresponding to *true positive, true negative, false positive* and *false negative*. These values are used for computing the sensitivity, specificity, precision and F_1-scores. A *true positive* alignment is a true (or real) alignment between a pair of reads which is correctly identified by the program. A *true negative* alignment is a false (or non existing) alignment which is correctly rejected i.e. not reported by the program. A *false positive* alignment is a false alignment which is incorrectly reported by the program and a *false negative* alignment is a real alignment which is not reported by the program.

4.4.1 Compute Dynamic Programming Alignment

As explained in the previous section, the program reported alignments were compared against optimal alignments computed using Smith-Waterman algorithm. The original code in *EstimateROC* utility used Smith-Waterman for some specific cases. For example, during specificity computation, if a program reported alignment is missing from BLASR mapping, dynamic programming is used to evaluate the correctness of the alignment. This is done only if the user has enabled dynamic programming option. We use a modified version of the code which has stricter checks for lengths and bounds. The modified function *ComputeDP* first computes an optimal alignment, A_{opt}, between two reads using the Smith-Waterman dynamic programming algorithm. It is ensured that this is a good alignment as per the evaluation criterion, otherwise this alignment is ignored. Next, we check if the program has reported a corresponding alignment. If no alignment is found, it is considered as a missing alignment in the program output. The Algorithm 4.4.1 provides the details of checks performed with respect to optimal alignment. Assume that the program reported an alignment, A_{rep}. We then compare the length, direction and bounds of the alignment reported by the program with those of the optimal alignment. This step is essential for correctness because the program could report a very different alignment between the same reads. The use of an optimal alignment algorithm increases the accuracy of evaluation.

Algorithm 4.4.1: COMPUTEDP(r_1, r_2, A_{rep})

$A_{opt} \leftarrow$ optimal alignment between r_1 and r_2
if ($A_{opt}.length < minOverlapLength$) **or**
($A_{opt}.score < minOverlapScore$)
  then return (bad_A_{opt})
if ($A_{rep} = void$)
  then return (no_A_{rep})
if ($A_{opt}.direction \neq A_{rep}.direction$) **or**
($|A_{opt}.length - A_{rep}.length| < 0.3A_{opt}.length$) **or**
($|A_{opt}.left - A_{rep}.left| > 0.3A_{opt}.length$) **or**
($|A_{opt}.right - A_{rep}.right| > 0.3A_{opt}.length$)
  then return (bad_A_{rep})
return ($good\_A_{rep}$)

The three functions used for evaluation, *EstimateSensitivity, EstimateSpecificity* and *EstimatePrecision*, are modified to correspond with our new *ComputeDP* function. The evaluation performed with our method is more accurate and strict therefore all programs exhibit a decline in performance. The Results section contains a comparison of several evaluation procedures.

4.4.2 Sensitivity computation

Sensitivity is a standard statistical measure which computes true positive rate or the positive cases which are correctly identified in a given sample of data. Statistically, it is computed as $TP \div (TP + FN)$. The Algorithm 4.4.2 gives details of the sensitivity computation. Note that in the algorithm, we account for a given alignment only if there exists a valid optimal alignment

between pair of reads.

Algorithm 4.4.2: ESTIMATESENSITIVITY( )

for $i \leftarrow 1$ **to** *numTrials*

do
- Pick random overlap from BLASR reference mapping.
- Assume the reads are r_1 and r_2.
- **if** (ComputeDP(r_1, r_2, A_{rep}) $\neq$ bad_A_{opt})
  - **then**
    - **if** (ComputeDP(r_1, r_2, A_{rep}) = $good\_A_{rep}$)
      - **then** $TP \leftarrow TP + 1$
      - **else** $FN \leftarrow FN + 1$

return $(\frac{TP}{TP+FN})$

4.4.3 Specificity computation

Specificity measures the true negative rate or all the negative cases which are correctly identified in a given sample of data. Statistically, it is computed as $TN \div (TN + FP)$. The Algorithm 4.4.3 gives details of the specificity computation. We start by randomly generating two sequence ids. The optimal alignment is computed between the sequences. Depending on optimal alignment and program output, it is categorized as false positive or true negative.

Algorithm 4.4.3: ESTIMATESPECIFICITY( )

for $i \leftarrow 1$ **to** *numTrials*

do
- Generate two random read IDs: r_1 and r_2.
- **if** (overlap $A_{rep}(r_1, r_2)$ exists in program output)
  - **then**
    - **if** (ComputeDP$(r_1, r_2, A_{rep}) \neq good_A_{rep}$)
      - **then** $FP \leftarrow FP + 1$
  - **else**
    - **if** (ComputeDP$(r_1, r_2, A_{rep}) = bad_A_{opt}$)
      - **then** $TN \leftarrow TN + 1$

return $(\frac{TN}{TN+FP})$

4.4.4 Precision computation

Precision is the measure of consistency of results when an experiment is repeated certain number of times. Statistically, it is defined as the fraction of true alignments among the total alignments reported by the program or $TP \div (TP + FP)$. Similar to other algorithms, decision of correctness is based on the optimal alignment reported by the Smith-Waterman algorithm. Algorithm 4.4.4 provides details of the precision computation.

Algorithm 4.4.4: ESTIMATEPRECISION( )

for $i \leftarrow 1$ **to** *numTrials*

do
- Pick random alignment $A_{rep}(r_1, r_2)$ from program output.
- **if** (ComputeDP$(r_1, r_2, A_{rep}) = good_A_{rep}$)
  - **then** $TP \leftarrow TP + 1$
  - **else** $FP \leftarrow FP + 1$

return $(\frac{TP}{TP+FP})$

4.4.5 F_1 score computation

The statistical measures such as precision and recall (same as sensitivity) give us a very good understanding of the program's characteristics. However, it is possible that a program depicts a characteristic of low sensitivity, high specificity and a very high precision at the same time. For genome alignment, such cases may not give a clear indication of correctness and accuracy of the algorithm.

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

To counter such cases, we decided to compute one more statistical measure, called F_1 *score*. F_1 score is a measure of accuracy and it is defined as the harmonic mean of the precision and the recall for a given sample of data. The measure is an average of the two when precision and recall are very close.

4.5 Results

HISEA is evaluated against the most popular programs for PacBio read alignment: BLASR [16], DALIGNER [44], GraphMap [57], MHAP [6] and Minimap [35]. Also, we assessed the performance of HISEA for assembling PacBio data by including HISEA in the Canu assembly pipeline [33] and compared it with the assembly generated using MHAP as the aligner in the same pipeline.

All the programs were run according to their own developers' suggestions as provided in their respective publications or websites. Minimap and DALIGNER were run as suggested. We tested BLASR with default parameters and the results were very poor. BLASR program's sensitivity is primarily affected by the *bestn* parameter which controls the number of alignments to be reported for each read. To get the highest sensitivity *bestn* parameter is tweaked for each dataset. Our test indicates *bestn* values similar to what has been used by MHAP. Hence BLASR was run with exactly same parameters as in MHAP paper. GraphMap was run with default

parameters as the only choice available in overlapping mode. MHAP was run with default parameters, except the number of hashes, which was set to 1256, instead of the default 512, for increased sensitivity. Minimap was run with window size 5 (default is 10), as recommended by the designers. MHAP and Minimap were also tested on varying number of hashes and window sizes respectively. The results for these tests are shown in Table 4.6 and 4.7 respectively. HISEA was run with default parameters. All tests were performed on a DELL PowerEdge R620 computer with 12 cores Intel Xeon at 2.0 GHz and 256 GB of RAM, running Linux Red Hat, CentOS 6.3.

The datasets have been downloaded from Pacific Biosciences DevNet Datasets (https://github.com/PacificBiosciences/DevNet/wiki/Datasets). The datasets used for this evaluation are given in Table 4.2.

Table 4.2: SMRT datasets used in for evaluation

Genome	Reference number	Coverage	Chemistry	Genome size (Mbp)
E.coli	NC_000913	85x	P5C3	4.64
S.cerevisiae	NC_001133.9	117x	P4C2	12.16
C.elegans	WS222	80x	P6C4	100.2
A.thaliana	TAIR10	110x	P4C2	134.6
D.melanogaster	Ref v5	90x	P5C3	129.7

4.5.1 Alignment results

Standalone comparison

It is very time consuming to test full datasets for all possible alignments. So, to assess the quality of the competing programs, a subset of 1Gbp dataset was created by randomly sampling reads from the original datasets. The two smallest genomes *E.coli* and *S.cerevisiae*, use full datasets since they are close to 1Gbp with the given coverage. The sensitivity, specificity, precision and F_1 score values for all programs are given in Table 4.3. They were computed using the *EstimateSensitivity, EstimateSpecificity* and *EstimatePrecision* procedures that we

described in the Section 4.4. The values for sensitivity, specificity, precision and F_1-score are given in percentages. A dash for a value means that the program crashed with segmentation fault. The coverage corresponding to 1Gbp dataset is also shown in parenthesis along with the dataset name. The best value for a test is shown in bold and is represented by a darker green color. At the bottom of the table, the average values are computed for each program from corresponding results for each dataset in the table.

Table 4.3: Comparison for the 1 Gbp datasets.

Genome (Coverage)	Parameter	BLASR	DALIGNER	GraphMap	MHAP	Minimap	HISEA
E.coli (85x)	Sensitivity	96.44	76.26	41.03	83.74	91.80	**97.06**
	Specificity	98.61	99.82	99.83	99.90	**99.93**	99.87
	Precision	**98.21**	83.56	39.66	97.15	97.13	97.95
	F_1-score	97.32	79.74	40.33	89.95	94.39	**97.50**
S.cerevisiae (117x)	Sensitivity	21.72	–	4.41	62.08	9.35	**91.70**
	Specificity	99.61	–	99.95	99.77	**99.98**	99.77
	Precision	**96.25**	–	45.48	89.29	94.30	94.31
	F_1-score	35.44	–	8.04	73.24	17.01	**92.99**
C.elegans (10x)	Sensitivity	92.71	73.96	36.21	80.43	85.38	**93.38**
	Specificity	98.62	99.97	99.98	99.97	**99.98**	99.97
	Precision	**93.23**	78.24	38.65	45.46	89.80	88.83
	F_1-score	**92.97**	76.04	37.39	58.09	87.53	91.05
A.thaliana (8x)	Sensitivity	6.69	64.13	10.83	76.19	23.55	**90.89**
	Specificity	99.99	99.86	**99.97**	99.91	**99.97**	99.89
	Precision	**98.72**	83.09	47.98	88.78	84.00	94.58
	F_1-score	12.53	72.39	17.67	82.00	36.79	**92.70**
D.melanogaster (8x)	Sensitivity	40.69	68.85	17.76	71.86	40.72	**91.40**
	Specificity	99.90	99.90	99.98	99.94	**99.99**	99.93
	Precision	**94.14**	77.45	37.76	72.47	83.93	90.10
	F_1-score	56.82	72.90	24.16	72.16	54.84	**90.75**
Average	Sensitivity	51.65	70.80	22.05	74.86	50.16	**92.89**
	Specificity	99.35	99.89	99.94	99.90	**99.97**	99.89
	Precision	**96.11**	80.59	42.72	78.63	89.83	93.15
	F_1-score	59.02	75.27	18.04	75.09	58.11	**93.00**

The mandatory inputs to *EstimateROC* are the reference genome, the reads, minimum alignment length, number of trials and the mapping of the reads to the reference. The mapping

of the reads to the reference is computed using the BLASR program. It is used as a guidance to pick random reads with possible overlaps for evaluation as explained in Section 4.4. The minimum alignment length is set to 2,000 bps and all trials were conducted with a sample of 50,000 alignment with no repetition.

HISEA has the best sensitivity for all datasets. On average, it has 16% higher sensitivity than the second best program - MHAP. Specificity is high for all the programs and it is more than 99% for all of them. Comparing absolute values of specificity indicate that Minimap has the highest specificity for all datasets but at the same time it has very low sensitivity. BLASR has the highest precision but again its sensitivity values vary a lot for each dataset and on an average it has a low sensitivity. HISEA has the highest sensitivity and it comes a close second for precision. It is not clear if one should use a program with best sensitivity or best precision. To better understand the relationship between them, we have computed the F_1-scores (defined in Section 4.4.5), also shown in Table 4.3. The F_1-score for HISEA is much higher than all the other programs which makes it a clear winner. The F_1 scores for DALIGNER and MHAP fall behind by 18% and 19% respectively. The remaining three programs BLASR, Minimap and GraphMap have F_1 scores of 59.02%, 58.11% and 18.04% respectively which are very low as compared to HISEA.

The time and memory comparison for the same 1Gbp datasets is presented in Table 4.4. Minimap and GraphMap are clearly the fastest and BLASR the slowest. HISEA is in the middle, behind MHAP and DALIGNER. Space-wise, Minimap is again the best, followed closely by BLASR. HISEA comes third, with GraphMap following behind it. MHAP and DALIGNER used the most memory. MHAP is implemented in JAVA which generally requires more memory because of the *Java Virtual Machine*. The java command-line parameter -Xmx is used to set the maximum heap size for MHAP stand alone invocation. The default maximum java heap size depends on the platform and the amount of memory in the system. For our systems, the default heap size was not sufficient to perform the tests. We set -Xmx parameter to 200G which was sufficient for all tests but it does not capture true overlapper memory for

MHAP. The reported memory usage for MHAP is a sum of the overlapper memory and the memory required for *Java Virtual Machine* environment. The best values are in bold and a dash means a segmentation fault in program output. The time is reported in CPU hours and memory is in GB.

Table 4.4: Time and memory comparison for the 1 Gbp datasets.

Genome	Time (h) Memory (GB)	BLASR	DALIGNER	GraphMap	MHAP	Minimap	HISEA
E.coli	Time	113.0	3.0	0.3	3.0	**0.1**	4.0
	Memory	**7.1**	124.6	42.3	210.0	8.8	25.5
S.cerevisiae	Time	283.2	–	0.6	10.6	**0.3**	23.5
	Memory	**13.3**	–	71.0	210.0	15.1	56.5
C.elegans	Time	333.6	4.1	0.6	4.3	**0.2**	23.6
	Memory	14.5	248.2	59.0	210.0	**9.8**	46.4
A.thaliana	Time	43.2	8.1	0.6	5.9	**0.2**	12.2
	Memory	10.3	248.2	60.0	210.0	**9.9**	45.3
D.melanogaster	Time	355.2	12.5	0.4	4.8	**0.1**	95.1
	Memory	16.7	204.2	59.0	210.0	**9.7**	48.1

Sensitivity - a deep dive

It was observed that our sensitivity numbers are lower when compared with original *Estimate-ROC* utility from Berlin *et al.* [6]. We expected it as we added evaluation checks which are stricter than those of Berlin *et al.* [6]. We wanted to understand the performance of each program when these checks are relaxed. Table 4.5 shows results for four different ways of evaluating the sensitivity under relaxed criterion. The strictest check is one which is used in our evaluation and involves checking of precise bounds of all alignments - in the table this is labelled as “bounds”. This condition is relaxed in our next sensitivity test and it is referred to in the table as “length” - which only checks for length of the alignment. This condition is further relaxed to simply check for the presence of the alignment and it is referred to in the table as “presence”. This is our weakest test and we expected all programs to perform well under this criterion. However, the results in Table 4.5 do not validate our hypothesis except

for DALIGNER, MHAP and HISEA. Further, to be fair with the evaluation, we also include the results from original *EstimateROC* utility and it is referred as " Berlin *et al.*" in the table. While there are differences among all these sensitivity modes, HISEA clearly remains at the top, followed by DALIGNER and MHAP. The other three programs perform very poorly. It is interesting to note the very high sensitivity of DALIGNER in the "presence" only scenario. This indicates that even though many alignments were found by DALIGNER, their length and bounds are not close to optimal.

Table 4.5: Comparison of several types of sensitivity computations on the 1 Gbp datasets.

Genome	Sensitivity	BLASR	DALIGNER	GraphMap	MHAP	Minimap	HISEA
E.coli	presence	98.99	**99.57**	83.49	85.00	97.12	99.12
	length	96.60	76.68	70.52	83.92	91.95	**97.20**
	bounds	96.44	76.26	41.03	83.74	91.80	**97.06**
	Berlin et al.	97.35	80.41	84.74	84.82	94.11	**97.86**
S.cerevisiae	presence	22.60	–	9.27	67.43	10.31	**98.02**
	length	21.96	–	4.63	63.93	9.35	**93.93**
	bounds	21.72	–	4.41	62.08	9.35	**91.70**
	Berlin et al.	29.70	–	12.06	71.92	13.23	**91.82**
C.elegans	presence	98.99	**99.68**	75.91	87.37	93.39	98.97
	length	96.64	78.08	59.37	85.40	88.47	**97.01**
	bounds	92.71	73.96	36.21	80.43	85.38	**93.38**
	Berlin et al.	92.48	78.59	73.02	79.77	87.04	**92.60**
A.thaliana	presence	6.83	92.11	21.21	85.69	29.76	**99.15**
	length	6.72	69.09	12.21	83.14	24.00	**96.84**
	bounds	6.69	64.13	10.83	76.19	23.55	**90.89**
	Berlin et al.	11.11	65.52	14.17	60.86	17.99	**74.71**
D.melanogaster	presence	42.59	97.74	36.48	82.53	47.48	**98.25**
	length	41.26	71.77	21.74	76.69	41.80	**94.38**
	bounds	40.69	68.85	17.76	71.86	40.72	**91.40**
	Berlin et al.	41.67	75.62	35.94	78.19	43.36	**89.81**
Average	presence	54.00	97.28	45.27	81.60	55.61	**98.70**
	length	52.64	73.91	33.69	78.62	51.11	**95.87**
	bounds	51.65	70.80	22.05	74.86	50.16	**92.89**
	Berlin et al.	54.46	75.04	43.99	75.11	51.15	**89.36**

Sensitivity vs overlap size

All the programs used in this evaluation produce approximate alignment bounds based on some heuristics. Since the evaluation allows an error of around 30% in length and bounds, it is easier to find long overlaps with correct bounds as compared to short overlaps. To understand the impact on sensitivity performance of each program with increasing overlap length, the aligners sensitivity is plotted as a function of mean overlap length. The mean overlap is computed for a window of size 1000 base pairs. For example, the first point represents a mean value for a window of overlap length 1001-2000, second point is a mean value for a window of size 2001-3000 and so on.

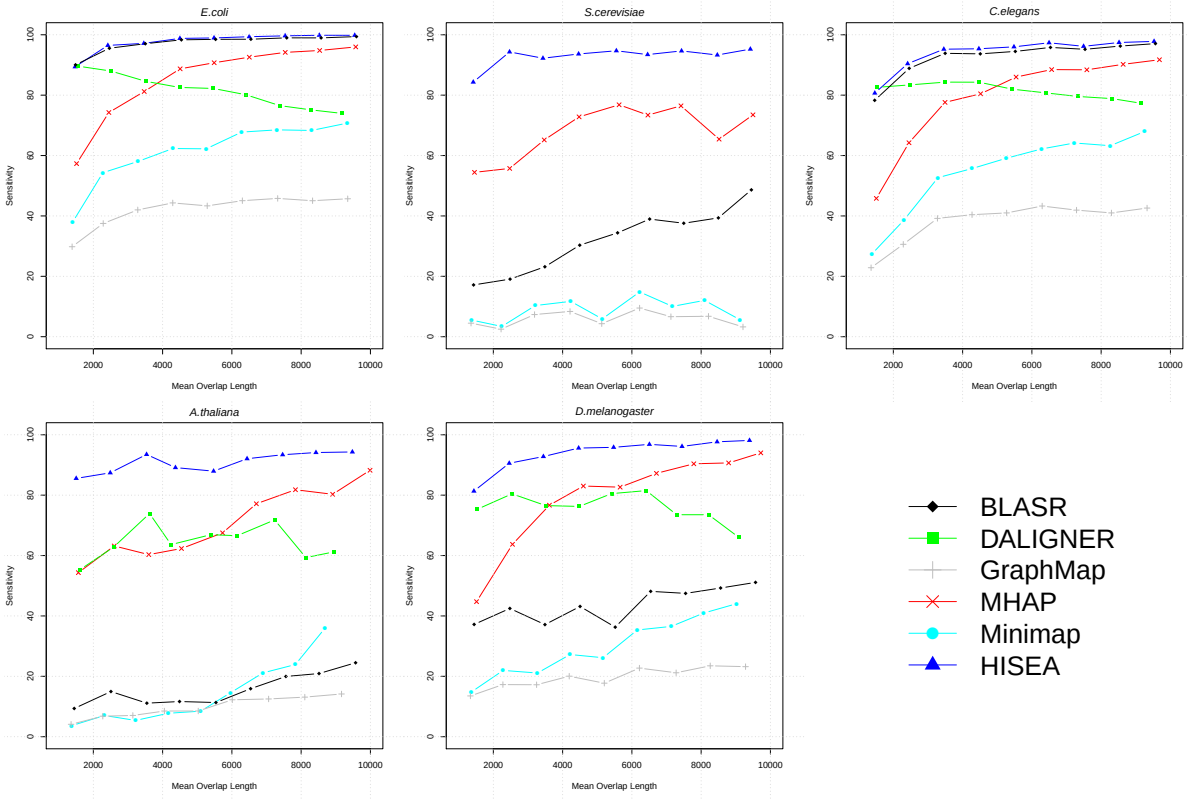

Figure 4.8: Sensitivity as a function of mean overlap length.

Figure 4.8 shows plots for all the datasets. As expected, the sensitivity increases with the overlap length for all aligners except DALIGNER. The sensitivity of HISEA remains very high for both short and long overlaps and it improves with longer overlap lengths. MHAP

shows a similar trend but its sensitivity for short overlaps is very low. BLASR, Minimap and GraphMap seem to have been optimized for more recent pacbio chemistries. Their result are better for newer chemistries compared to their own results on older ones. They perform poorly on oldest chemistry P4C2 datasets.

MHAP sketch size and Minimap minimizers

Both MHAP and Minimap can have their parameters adjusted to improve sensitivity. Again, to be fair with these programs, we investigated the effect of changing these parameters on sensitivity.

MHAP is based on a technique called MinHash [12] in order to compute the overlaps. MinHash reduces any sequence into a fixed number of fingerprints, called a sketch. The number of fingerprints in a sketch depends on the number of hash functions used. Once a sequence is converted into a sketch, the entire sequence is represented by it and the sequence is never referenced directly again. This makes all the operations related to sequences very efficient. However, in this process, some sequence information is lost. Hence, it is expected that larger sketch size (meaning more hash functions) will better capture sequence information, but it impacts the processing time negatively. Based on this observation, it is clear that using a larger sketch size increases the sensitivity at the cost of run time performance.

MHAP is one of the fastest aligner and it is worth investigating the effect of this parameter. The sensitivity results in Table 4.3 used a sketch size 1256 instead of the default 512, for improved sensitivity. Table 4.6 shows the results for increasing sketch size with increments of 512, starting from 1256. The sensitivity increases slightly but never comes close to HISEA. Also, precision decreases and so the F_1-score increases very little (or decreases dramatically, as it happens for *C.elegans*). The running time increases up to 10 times when changing sketch size from 1256 to 3816. Overall, increasing the sketch size is clearly not improving the performance of MHAP. Note that the results for the first column (sketch size 1256) appear also in Table 4.3. They are repeated here for the convenience of comparison.

Table 4.6: Effect of increasing sketch size on MHAP sensitivity.

Genome	Parameter	MHAP *skecth* size					
		1256	1768	2280	2792	3304	3816
E.coli	Sensitivity	83.74	85.75	86.52	86.87	87.05	87.16
	Specificity	99.90	99.86	99.84	99.82	99.81	99.80
	Precision	97.15	96.99	96.82	96.89	96.88	97.70
	F_1-score	89.95	91.02	91.38	91.61	91.70	92.13
S.cerevisiae	Sensitivity	62.08	63.67	64.32	64.52	64.62	64.69
	Specificity	99.77	99.72	99.66	99.63	99.58	99.56
	Precision	89.29	88.79	88.69	88.62	88.55	88.30
	F_1-score	73.24	74.16	74.56	74.67	74.72	74.67
C.elegans	Sensitivity	80.43	81.81	82.37	82.62	82.69	82.73
	Specificity	99.97	99.93	99.90	99.88	99.85	99.82
	Precision	45.46	35.71	29.32	25.80	23.75	22.13
	F_1-score	58.09	49.72	43.25	39.32	36.90	34.92
A.thaliana	Sensitivity	76.19	77.05	77.38	77.49	77.55	77.57
	Specificity	99.91	99.87	99.86	99.85	99.84	99.83
	Precision	88.78	88.50	88.68	88.35	88.55	88.33
	F_1-score	82.00	82.38	82.65	82.56	82.69	82.60
D.melanogaster	Sensitivity	71.86	73.36	73.89	74.12	74.24	74.30
	Specificity	99.94	99.92	99.91	99.88	99.87	99.86
	Precision	72.47	72.00	72.07	72.46	71.45	71.62
	F_1-score	72.16	72.67	72.97	73.28	72.82	72.94

Similarly, the sensitivity of Minimap can be increased by using more minimizers. A minimizer is the smallest k-mer in a window of w consecutive k-mers. The default value is $w = 10$ but the recommended value by the designers for all-vs-all PacBio read self-mapping is $w = 5$ and this is what we have used in our tests. We have investigated the effect of increasing the number of minimizers by decreasing w. The results are presented in Table 4.7. The improvement is more significant for Minimap, but it starts from lower values. The improved performance is still far from HISEA, MHAP and DALIGNER. Note that the results for the first column ($w = 5$) appear also in Table 4.3. They are repeated here for the convenience of comparison.

Table 4.7: Effect of increasing number of minimizers on Minimap sensitivity.

Genome	Parameter	Minimap window size				
		5	4	3	2	1
E.coli	Sensitivity	91.80	93.08	94.13	95.24	96.29
	Specificity	99.93	99.92	99.93	99.92	99.91
	Precision	97.13	97.22	97.42	97.51	97.58
	F_1-score	94.39	95.10	95.75	96.36	96.93
S.cerevisiae	Sensitivity	9.35	9.64	9.94	10.36	11.00
	Specificity	99.98	99.98	99.97	99.97	99.97
	Precision	94.30	94.18	93.28	91.90	88.58
	F_1-score	17.01	17.49	17.97	18.62	19.57
C.elegans	Sensitivity	85.38	86.63	87.63	88.77	89.80
	Specificity	99.98	99.98	99.98	99.98	99.97
	Precision	89.80	89.77	89.05	88.11	85.76
	F_1-score	87.53	88.17	88.33	88.44	87.73
A.thaliana	Sensitivity	23.55	26.90	31.21	37.08	45.56
	Specificity	99.97	99.98	99.96	99.96	99.96
	Precision	84.00	84.77	85.48	86.43	87.94
	F_1-score	36.79	40.84	45.73	51.90	60.02
D.melanogaster	Sensitivity	40.72	42.82	45.51	49.11	54.00
	Specificity	99.99	99.98	99.98	99.98	99.97
	Precision	83.93	83.12	82.87	81.85	81.25
	F_1-score	54.84	56.52	58.75	61.39	64.88

4.5.2 Assembly results

The HISEA aligner has been integrated in the Canu assembly pipeline, which uses MHAP as the primary aligner. We refer to the new pipeline as Canu+HISEA and the original pipeline as Canu+MHAP. The assembly results are generated for the same five datasets mentioned in Table 4.2. We believe that better assembly is generated by alignments of high sensitivity and high precision. In the following section, we first compare the sensitivity of the alignment produced within the pipeline and then compare the assembly outcome.

Sensitivity of HISEA and MHAP - assembly pipeline

Table 4.8: Sensitivity, specificity, precision and F_1-score for HISEA and MHAP program output within the Canu pipeline.

Genome	Cov.	Sensitivity		Specificity		Precision		F_1-score	
		MHAP	HISEA	MHAP	HISEA	MHAP	HISEA	MHAP	HISEA
E.coli	30x	92.05	**95.31**	99.87	**99.94**	92.93	**98.65**	92.49	**96.96**
	50x	77.47	**93.95**	99.93	**99.94**	96.74	**98.59**	86.04	**96.21**
S.cerevisiae	30x	85.01	**92.94**	99.72	**99.77**	88.06	**94.24**	86.51	**93.59**
	50x	72.12	**94.07**	**99.89**	99.82	90.80	**94.29**	80.39	**94.18**
C.elegans	30x	82.64	**93.24**	99.80	**99.97**	7.21	**91.37**	13.26	**92.30**
	50x	67.59	**93.98**	**99.99**	99.98	76.53	**91.77**	71.78	**92.86**
A.thaliana	30x	80.94	**88.02**	99.88	**99.90**	87.42	**95.02**	84.06	**91.39**
	50x	57.06	**82.79**	**99.93**	99.90	89.46	**95.71**	69.68	**88.78**
D.melanogaster	30x	79.72	**91.82**	99.93	**99.95**	64.60	**90.25**	71.37	**91.03**
	50x	55.26	**89.59**	**99.97**	99.94	74.86	**90.52**	63.58	**90.05**
Averages	30x	84.07	**92.27**	99.84	**99.91**	68.04	**93.91**	69.54	**93.05**
	50x	65.90	**90.88**	**99.94**	99.92	85.68	**94.18**	74.29	**92.42**

MHAP uses an optimized set of options for generating alignments within the Canu assembly pipeline. This is different from running it in stand alone mode. It is therefore important to compare the sensitivity, specificity and precision of the alignment produced by the two programs before assembly outcome is compared. The assembly is generated for two different coverage levels: 30x and 50x. The 30x and 50x coverage datasets were sampled using the utility *fastqSample* available from the Canu pipeline [33]. It is still very expensive to get high coverage Pacbio sequencing data. Therefore, we aim to produce better assembly with smaller amount of coverage. Koren *et al.* [33] claims that Canu+MHAP pipeline reaches the best assemblies around 50x coverage, so this was considered as the higher bound of the coverage. We believe that 30x coverage data can produce assembly comparable to 50x coverage by using a better aligner.

The alignments computed by MHAP and HISEA while run in the Canu pipeline were extracted and analyzed in Table 4.8. HISEA has better sensitivity, precision and F_1 score

in all tests with very large differences for the 50x coverage datasets. The specificity of both programs is very high for all tests, with HISEA edging ahead for 30x coverage and MHAP for 50x. Similar to previous results, the best values are shown in bold and average is computed at the bottom of the table.

The assemblies produced by the two pipelines, Canu+MHAP and Canu+HISEA, have been evaluated using a modified version of our LASER program [29], which we discussed in Chapter 3. As explained, LASER computes many parameters for each assembly. The assembly results are shown in Table 4.9. We compare the number of contigs, NG50, the maximum contig size, the fraction of the genome covered by the assembly, the identity with the reference and the number of breakpoints (inversions, relocations and translocations). The Canu+HISEA pipeline has better values in 80% of the tests for the number of contigs, NG50, max contig size and genome fraction. Generally, the NG50 value for the Canu+HISEA assemblies is much larger than that of the Canu+MHAP. Canu+MHAP has fewer breakpoints than Canu+HISEA but the difference in number of breakpoints is usually small. Both pipelines have high identity with the reference. Overall, the assemblies computed by the Canu+HISEA pipeline are better. Moreover, the assemblies computed by Canu+HISEA for 30x coverage are comparable with those produced by Canu+MHAP for 50x coverage.

Table 4.9: Assembly comparison; Canu assembler is used with MHAP and HISEA as read aligners.

Genome	Parameter	Canu + MHAP		Canu + HISEA	
		30x	50x	30x	50x
E.coli	Contig #	**7**	3	8	**1**
	NG50	**2,771,323**	3,969,196	1,223,211	**4,642,165**
	Max contig	**2,771,323**	3,969,196	1,525,215	**4,642,165**
	% Ref	**99.85**	99.97	99.82	**100.00**
	Avg idy	**99.97**	**99.99**	**99.97**	**99.99**
	Breakpoints	**3**	**3**	**3**	**3**
S.cerevisiae	Contig #	43	31	**35**	**29**
	NG50	540,299	687,498	**682,168**	**774,485**
	Max contig	964,505	1,534,125	**1,537,586**	**1,534,133**
	% Ref	98.90	99.35	**99.12**	**99.58**
	Avg idy	99.81	**99.88**	**99.82**	**99.88**
	Breakpoints	**17**	**14**	**17**	**14**
C.elegans	Contig #	393	170	**127**	**133**
	NG50	636,401	1,987,017	**2,140,282**	**2,032,954**
	Max contig	2,648,207	4,224,025	**4,227,561**	**5,669,072**
	% Ref	96.00	**99.84**	**99.81**	99.80
	Avg idy	99.76	**99.91**	**99.85**	**99.91**
	Breakpoints	431	**423**	**390**	435
A.thaliana	Contig #	159	**99**	**140**	122
	NG50	3,331,858	6,715,370	**5,069,662**	**8,124,422**
	Max contig	**12,892,206**	14,177,369	12,890,806	**15,940,320**
	% Ref	92.22	**92.55**	**92.37**	92.51
	Avg idy	**99.17**	**99.22**	**99.17**	**99.22**
	Breakpoints	**2,550**	**2,693**	2,680	2,704
D.melanogaster	Contig #	597	390	**553**	**372**
	NG50	1,933,939	4,983,913	**6,417,268**	**13,672,005**
	Max contig	8,238,062	17,900,724	**17,366,974**	**25,767,672**
	% Ref	95.08	98.55	**96.47**	**98.65**
	Avg idy	**99.80**	**99.89**	**99.80**	99.87
	Breakpoints	**1,039**	**1,383**	1,254	1,461

The MHAP program is very fast and it makes the Canu+MHAP pipeline faster, as seen from the time values shown in Table 4.10. However, as noticed above, similar assemblies are produced by Canu+HISEA for 30x coverage, and those are always faster than those by

Canu+MHAP for 50x coverage. The memory consumption is always much lower for the Canu+HISEA pipeline. Note that in Table 4.10 the times are reported as wall clock times, since CPU times used by the overlapping programs are not available. Also, only the peak memory used by the entire assembly pipeline is available.

The *java* command-line parameter *-Xmx* is used to set the maximum heap size during MHAP invocation from the pipeline. The value of parameter *-Xmx* is set by *corMhapMemory* pipeline parameter which is user configurable. For this evaluation, the value of parameter *corMhapMemory* is set to 200 Gb for all datasets. The peak memory in each case is reported as 210 Gb. Similar configuration for Canu+HISEA pipeline uses much smaller memory footprint (less than 100 Gb) for all datasets.

The Canu+MHAP pipeline requires more memory in all cases, as seen from the space values shown in Table 4.10. The peak memory of this pipeline can be reduced by setting a smaller value for *corMhapMemory*. However, it impacts the overall assembly runtime. Similar behavior is expected in modified Canu+HISEA pipeline. To ensure unbiased evaluation, all parameter values are kept identical for both pipelines.

Table 4.10: Assembly time and space comparison.

Genome	Canu + MHAP				MHAP		Canu + HISEA				HISEA	
	30x		50x		30x	50x	30x		50x		30x	50x
	time	space	time	space	time	time	time	space	time	space	time	time
E.coli	**0.4**	210	**0.6**	210	**0.1**	**0.1**	**0.4**	**25**	0.7	**40**	**0.1**	**0.1**
S.cerevisiae	**1.1**	210	**2.0**	210	0.3	**0.4**	1.2	**63**	2.9	**76**	0.2	0.6
C.elegans	**24.5**	210	**59.6**	210	**2.4**	**2.5**	37.7	**83**	75.5	**82**	11.5	17.1
A.thaliana	**23.8**	210	**56.6**	210	**4.1**	**9.6**	42.3	**90**	98.0	**90**	15.3	35.0
D.melanogaster	**27.0**	210	**62.4**	210	**3.4**	**5.2**	51.8	**94**	112.8	**94**	19.7	33.6

We also generated mummer dot plots of all Canu+HISEA assemblies and corresponding reference genome from Table 4.2. A dot plot shows the relationship between the assembled contig and the reference genome. For all the dot plots, the assembled contig is on x-axis while the reference genome is on y-axis. A red dot indicates a forward-strand match, while a blue

dot indicates a reverse-complement match. The dotted vertical and horizontal lines indicate the contig and the chromosome boundaries. A diagonal indicates concordant matches while off-diagonal matches indicate assembly errors or differences with respect to the reference.

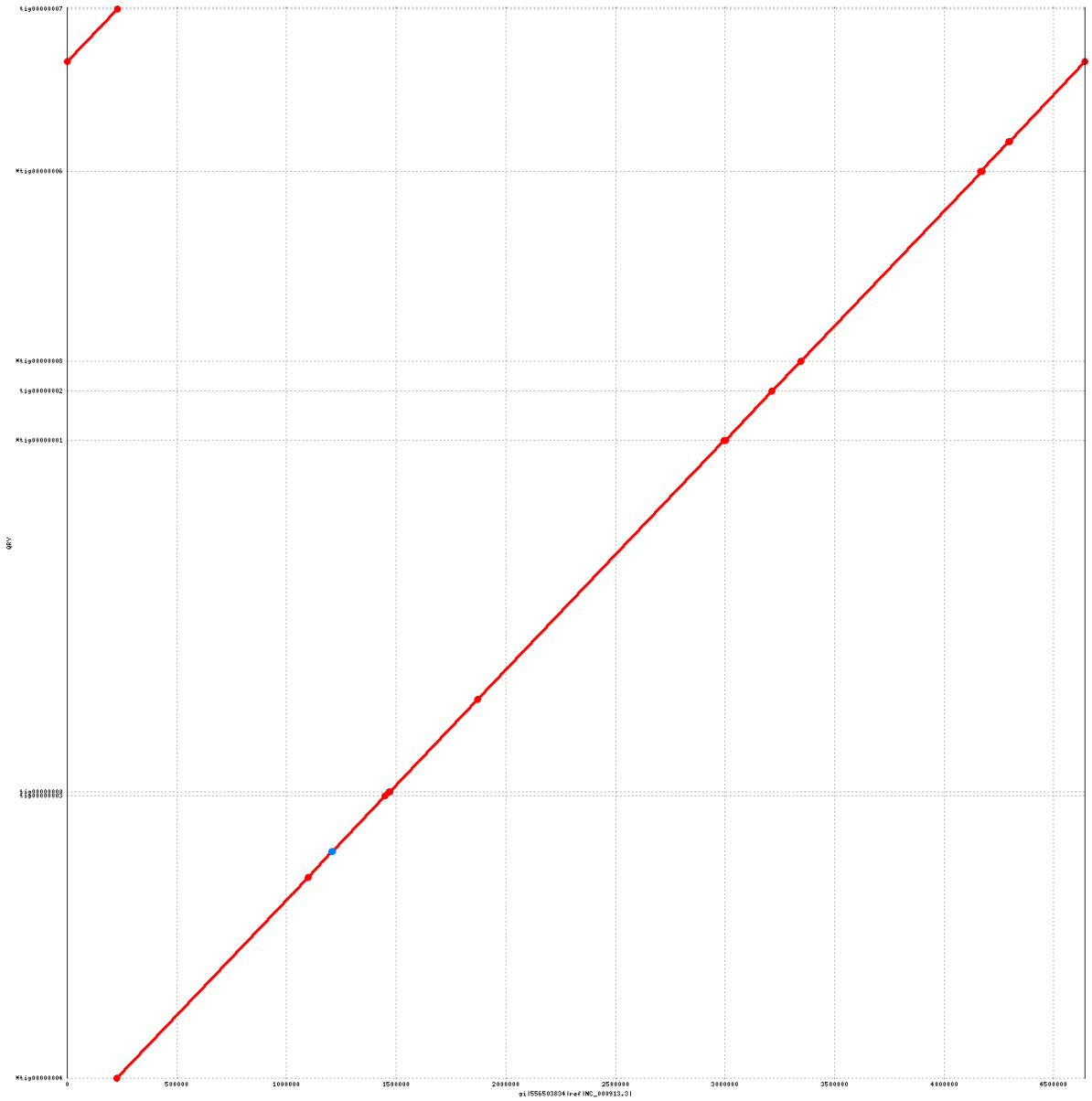

Figure 4.9: Mummer plot for *E.coli* 30x

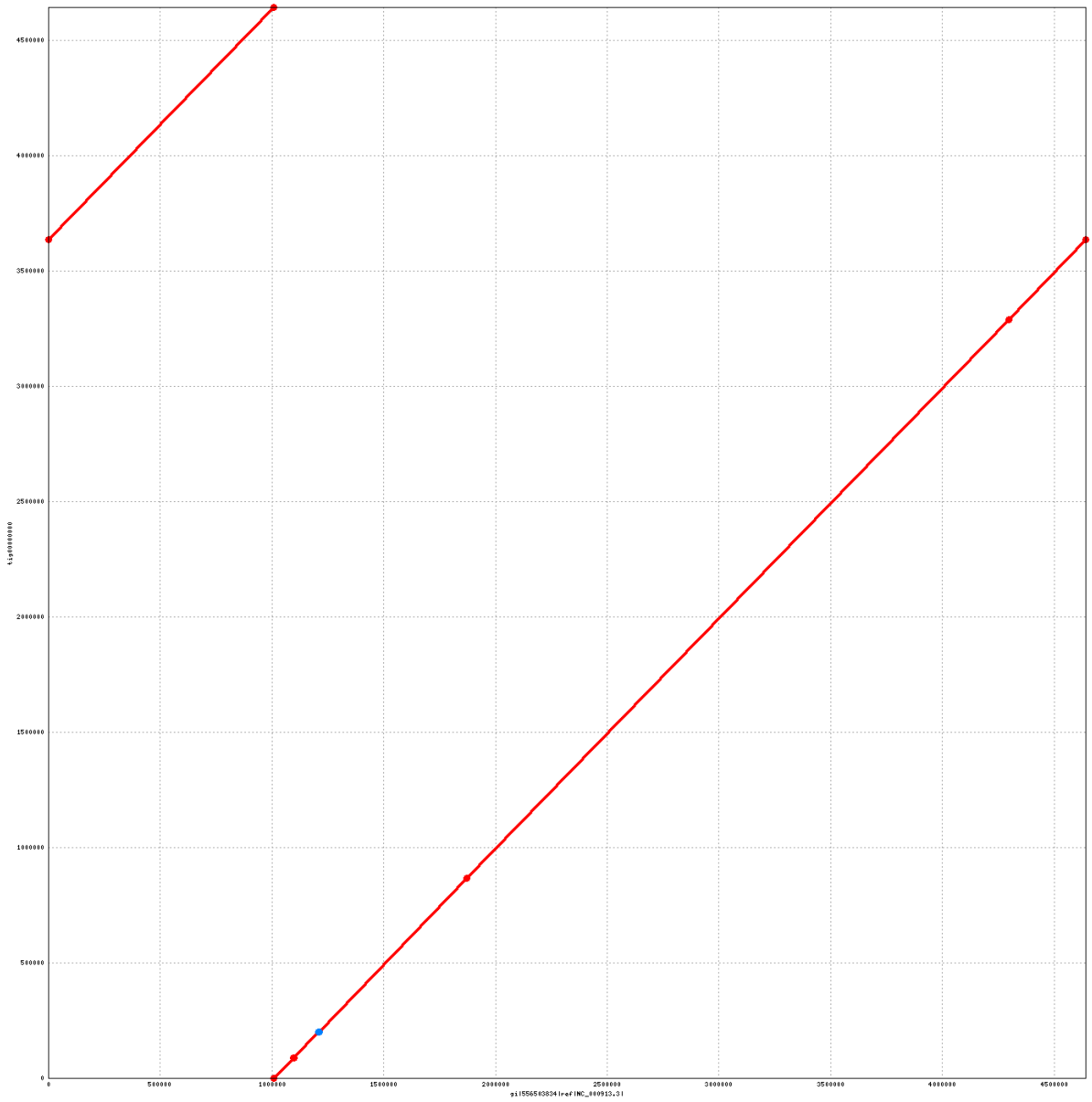

Figure 4.10: Mummer plot for *E.coli* 50x

Figures 4.9 and 4.10 shows the dot plots for *E.coli* 30x and 50x assemblies respectively. In both plots, a single contig matches the entire reference genome. The origin of the assembly is shifted due to the circular genome of *E.coli* and it is not an assembly error.

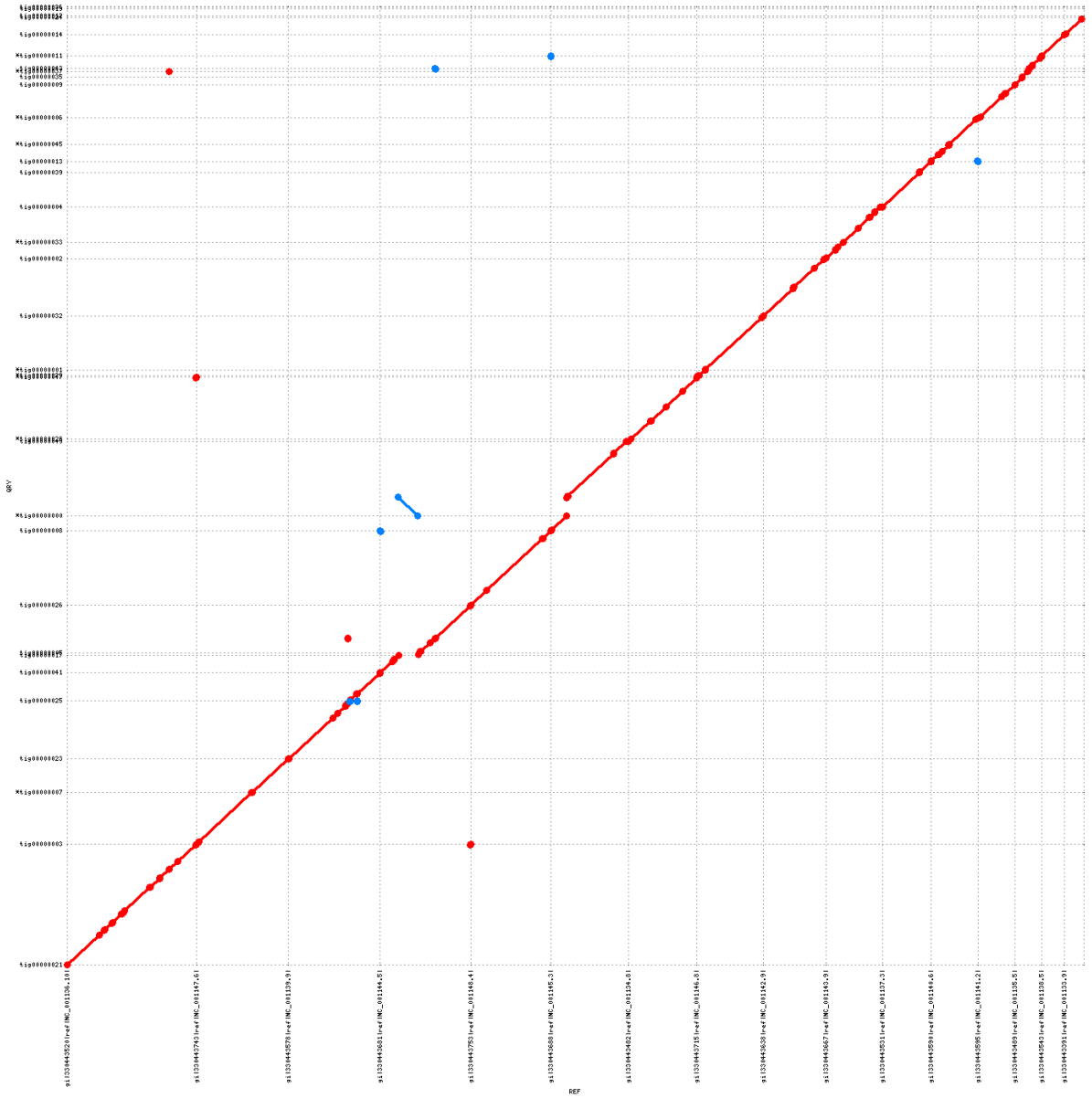

Figure 4.11: Mummer plot for *S.cerevisiae* 30x

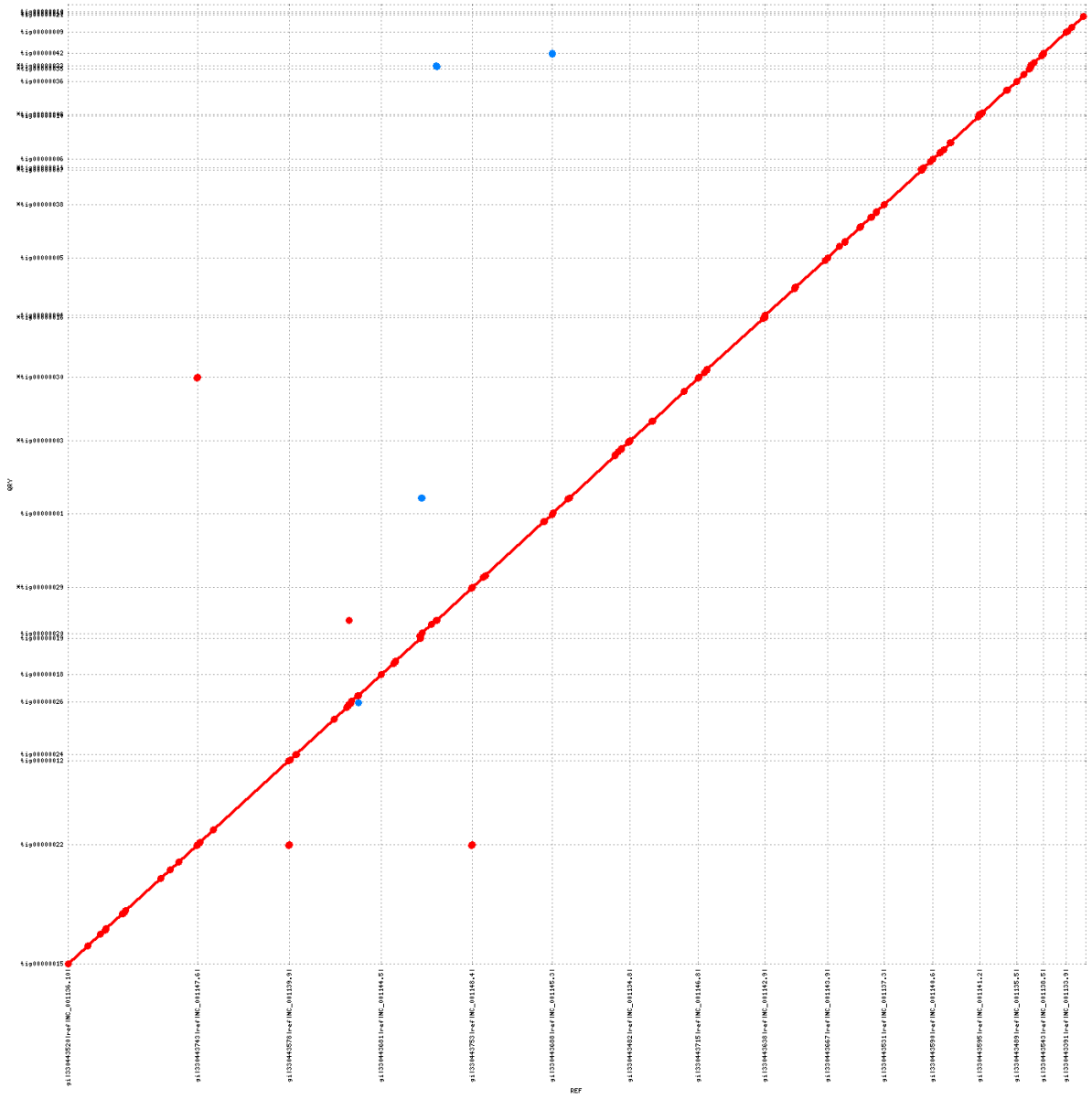

Figure 4.12: Mummer plot for *S.cerevisiae* 50x

Figures 4.11 and 4.12 shows the dot plots for *S.cerevisiae* 30x and 50x assemblies respectively. For both coverage levels, the majority of chromosomes are assembled into single contig. Clearly, 50x assembly is better than 30x as seen from the mummer plot.

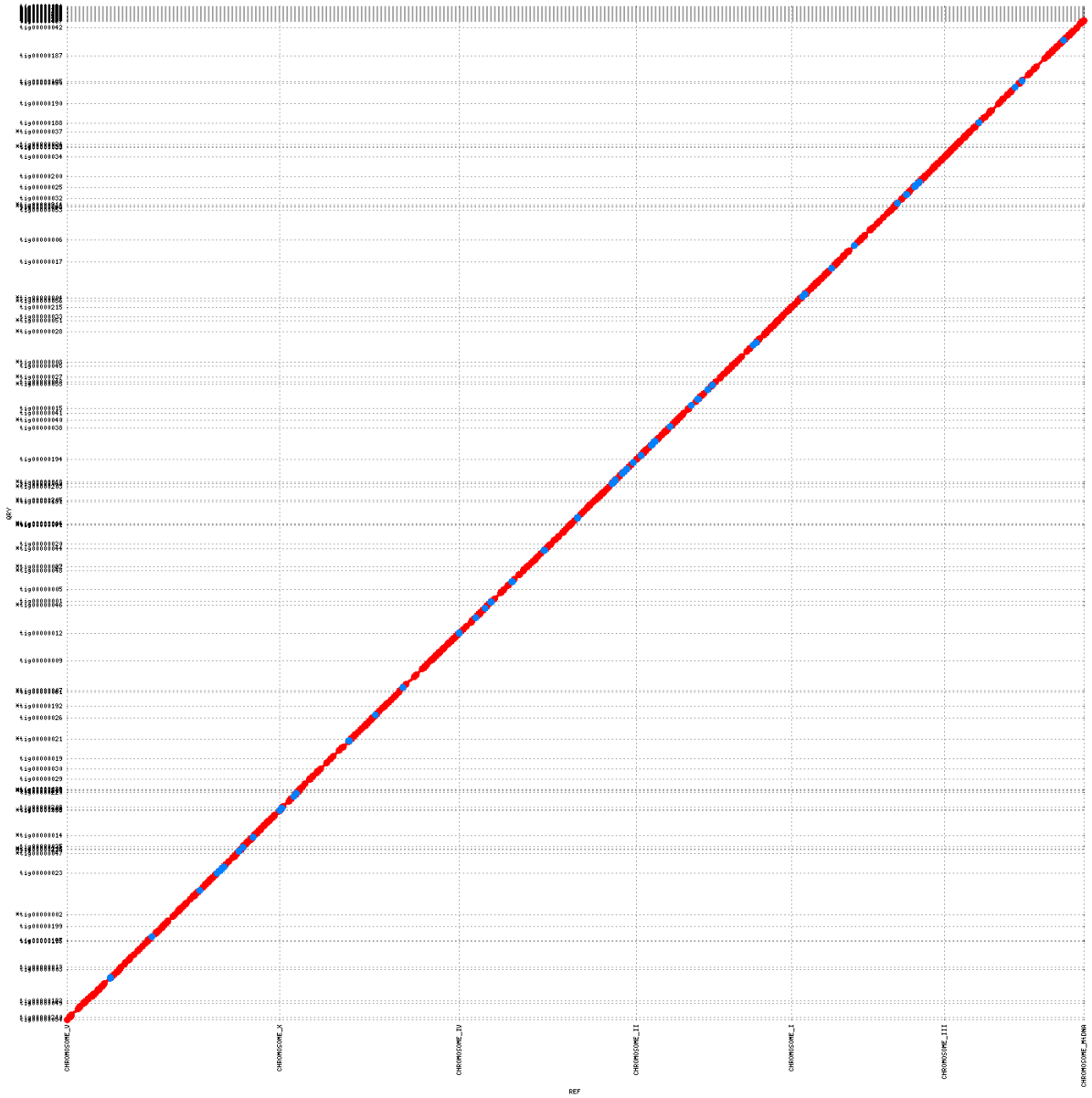

Figure 4.13: Mummer plot for *C.elegans* 30x

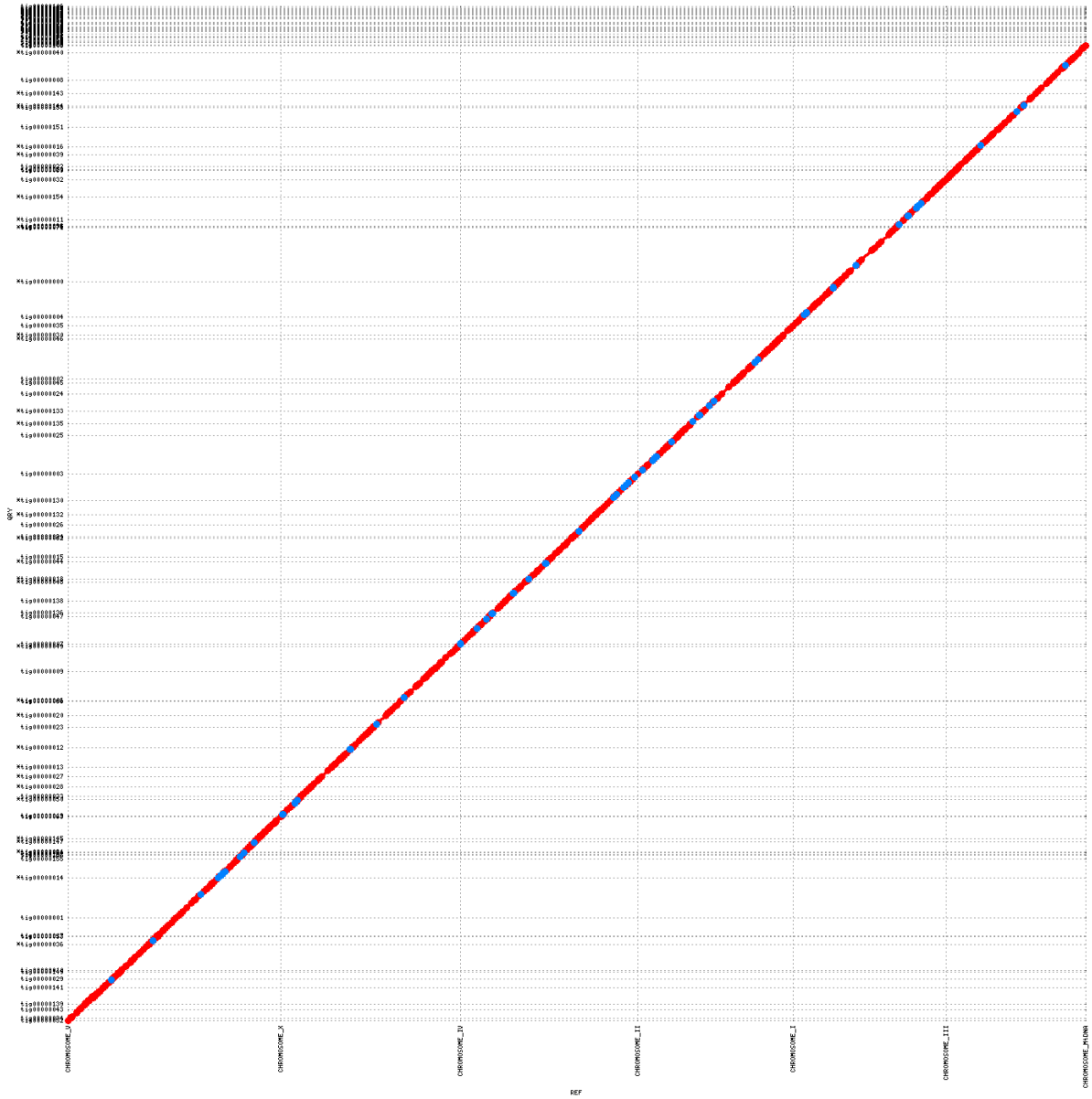

Figure 4.14: Mummer plot for *C.elegans* 50x

Figures 4.13 and 4.14 shows the dot plots for *C.elegans* 30x and 50x assemblies respectively. For both coverage levels, the assembly is equally good, as indicated by the assembly results in Table 4.9.

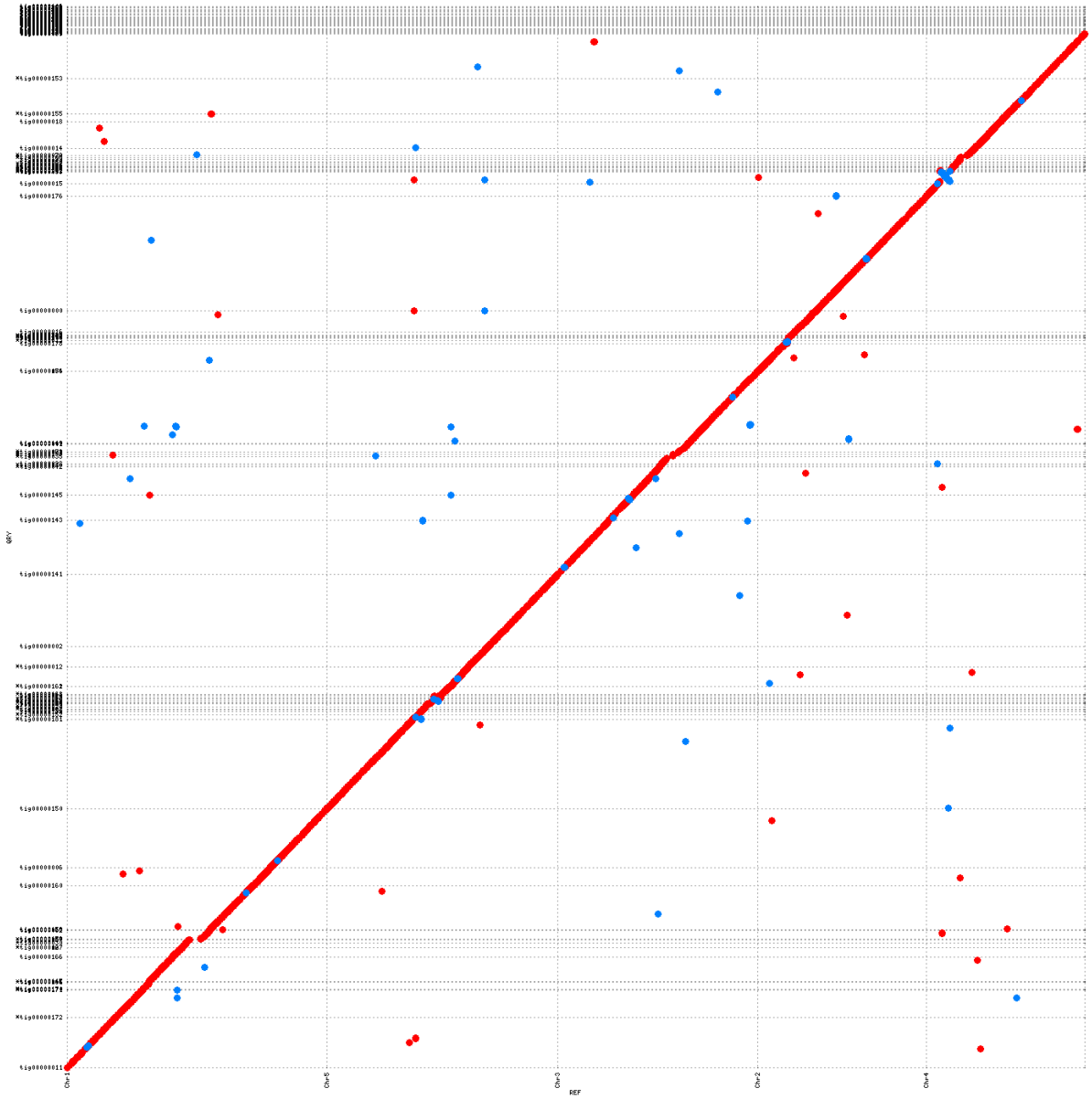

Figure 4.15: Mummer plot for *A.thaliana* 30x

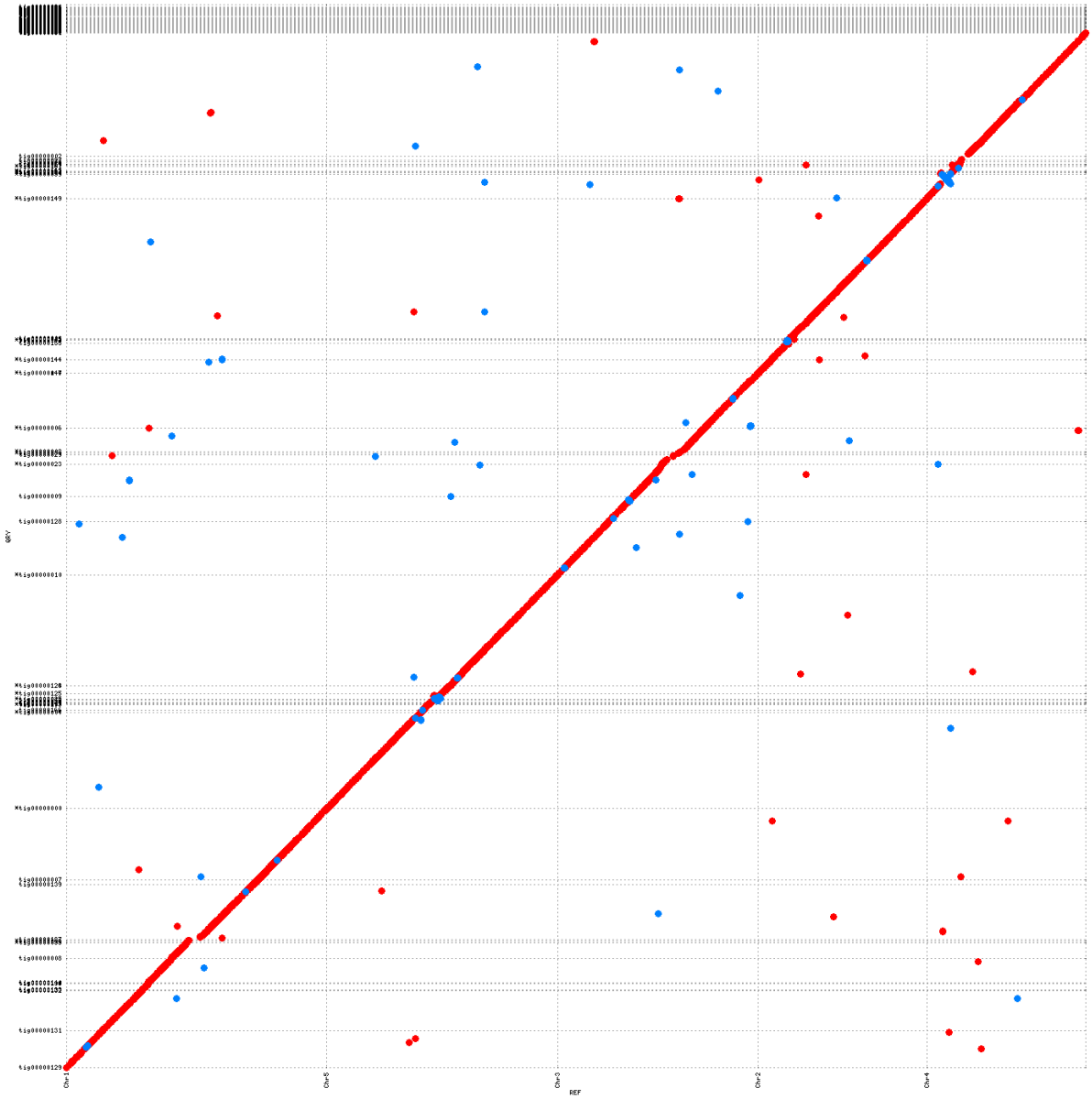

Figure 4.16: Mummer plot for *A.thaliana* 50x

Figures 4.15 and 4.16 shows the dot plots for *A.thaliana* 30x and 50x assemblies respectively. This is one of the difficult genomes to assemble. Although the assembled contigs are in many pieces, the plot show that they map correctly on the reference genome. A small inversion (blue line) is visible in the top right corner of the plot.

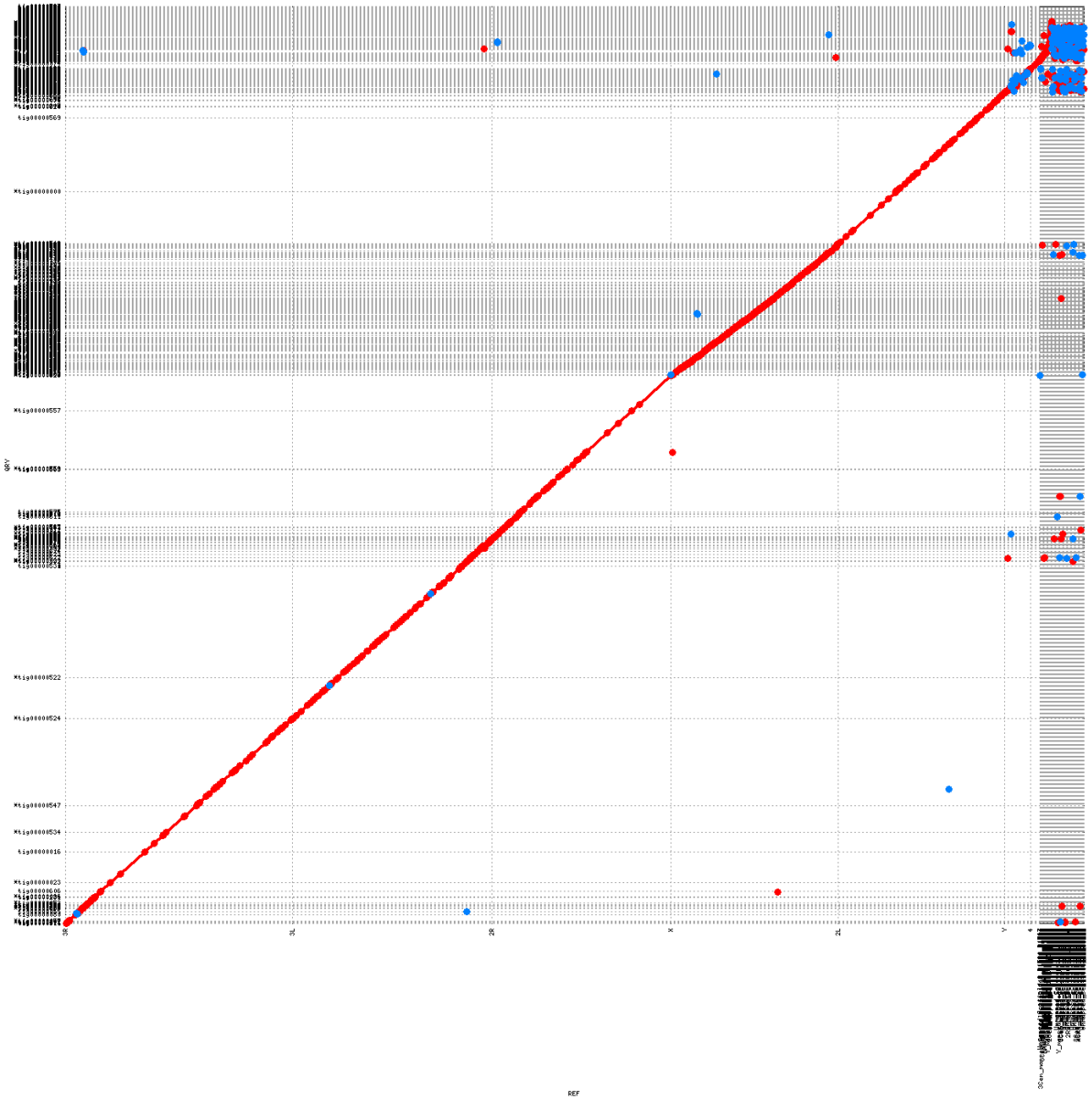

Figure 4.17: Mummer plot for *D.melanogaster* 30x

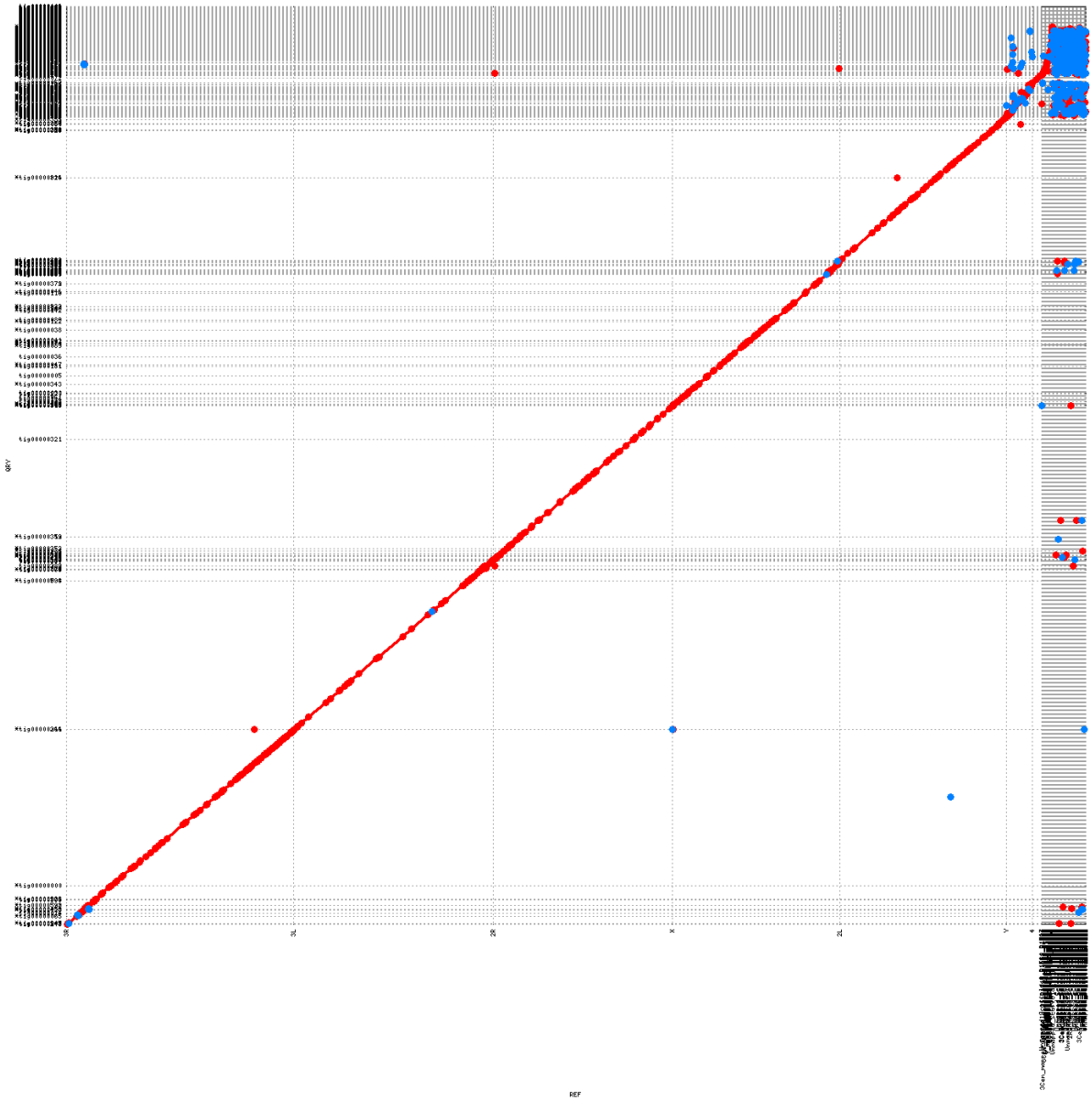

Figure 4.18: Mummer plot for *D.melanogaster* 50x

Figures 4.17 and 4.18 shows the dot plots for *D.melanogaster* 30x and 50x assemblies respectively. Some of the contigs represent full chromosomes. The highly fragmented region toward the top right corner is highly repetitive region and is not associated with a particular chromosome in the reference.

The HISEA program produces alignment output in M4 format [11] used by BLASR and MHAP. HISEA can also be integrated in other assembly pipelines, e.g., Miniasm [35] and

Falcon [17], by converting HISEA output to the format required by these pipelines.

4.6 Conclusions

Pacific Biosciences SMRT technology is a relatively new sequencing method that produces long but noisy reads. The aligners developed for previous sequencing methods do not perform well on this type of data. Our new HISEA algorithm for computing read alignments has introduced several new ideas, such as clustering of k-mer matches, estimating and filtering of matches based on error rate, and techniques for extending the alignments with shorter k-mer matches.

The HISEA algorithm currently produces alignments with highest sensitivity and comparable specificity with other algorithms. Integrated in the Canu pipeline [33], currently the best for assembling PacBio data, it produces better assemblies than Canu+MHAP. Moreover, the assemblies of Canu+HISEA at lower coverage, 30x, are comparable with those of Canu+MHAP at 50x coverage, while being faster and cheaper. We plan to modify HISEA in the future to work also with Oxford Nanopore sequencing technology [52].

Chapter 5

Conclusions and Future Research

5.1 Conclusions

We have addressed several fundamental problems in bioinformatics and provided significantly improved solutions. E-MEM is a new and efficient algorithm for computing MEMs between large genomes. It uses much lower memory and it is many times faster than other state-of-the-art MEM computation programs. The split parameter functionality of E-MEM significantly reduces the memory requirements. There is no theoretical upper bound on number of splits, which makes it unique and one of the most useful MEM computation program for very large genomes. E-MEM can be run as a stand alone program or a drop in replacement for any application where MEMs are used as seeds or anchor points.

LASER provides a practical solution for genome assembly evaluation problem. It inherits its functionality from QUAST, which is a leading program for genome and meta-genome assembly evaluation. It produces many metrics and visualizations for a thorough evaluation of genome assemblies. LASER provides significant improvement in terms of performance on QUAST, which is a bottleneck when it comes to evaluating very large genomes.

HISEA is a new algorithm for long read alignment, specifically designed for Pacific Biosciences SMRT sequencing data. HISEA is the most sensitive aligner as compared to others

and its specificity is comparable with other competing programs. The HISEA algorithm introduces new techniques for clustering, filtering and extension of *k*-mer matches which produce a highly sensitive alignment. Canu+HISEA pipeline produces better or comparable assemblies at 30x coverage than 50x coverage assemblies by Canu+MHAP pipeline. This makes it 40% cheaper which is significant given the high cost of Pacific Biosciences sequencing.

5.2 Future research

E-MEM is the fastest MEM computation program but there is still a scope for improving its performance for datasets with a large number of MEMs or the minimum MEM length is very small resulting in too many MEMs. The implementation relies on link list data structure for searching and eliminating redundant MEMs, which can become a bottleneck for above situations.

The core idea of E-MEM can be extended to find exact repeats or approximate repeats within a genome. This is an important problem in bioinformatics and it provides an insight into genomic evolution. The approach of E-MEM can be generalized to work with spaced seeds [38] in order to search efficiently for approximate matches. E-MEM has been developed as a stand alone program, however there is a scope to develop utilities around E-MEM for the purpose of read mapping and genome alignment.

Pacific Biosciences is a relatively new sequencing technology with very high error rate. HISEA can be used for the error correction of these long reads. HISEA algorithm produces alignments with highest sensitivity which is a key differentiator between HISEA and other state-of-the-art aligners. HISEA is currently integrated into Canu assembly pipeline, which is optimized for MAHP aligner. Having a new assembler specifically designed for HISEA will greatly improve the assembly results. HISEA algorithm can further be extended to support other long read sequencing technologies, most promising of which is Oxford Nanopore.

It would be interesting to enhance HISEA algorithm to align mixed sequencing data i.e. it

can be enhanced to align Illumina sequencing data with 2% error rate and Pacific Biosciences SMRT sequencing data with 15% error rate. This kind of alignment will have many applications. One such promising application would be to develop a hybrid genome assembler. The short reads from Illumina can be used for the construction of assembly graph. The graph simplification process can be guided by the alignment of long reads over short reads produced by HISEA. Aligning reads from two different sequencing technologies is a challenging problem and may require a significant changes. Currently, HISEA is a bit slower compared to MHAP and Minimap. There is also a scope for improving HISEA runtime performance.

Bibliography

[1] National human genome research institute - http://www.genome.gov/11006943.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[3] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Enhanced suffix arrays and applications. *Handbook of Computational Molecular Biology, Computer and Information Science Series*, 2006.

[4] Stephen F Altschul, Thomas L Madden, Alejandro A Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic acids research*, 25(17):3389–3402, 1997.

[5] Dmitry Antipov, Anton Korobeynikov, Jeffrey S McLean, and Pavel A Pevzner. hybridSPAdes: an algorithm for hybrid assembly of short and long reads. *Bioinformatics*, page btv688, 2015.

[6] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology*, 33(6):623–630, 2015.

[7] Pass NCEA Biology. Dna molecule structure, 2015. URL `http://www.passbiology.co.nz/biology-level-2/gene-expression`.

[8] Pacific Biosciences. Consensus accuracy, 2013. URL http://www.pacb.com/smrt-science/smrt-sequencing/accuracy/.

[9] Pacific Biosciences. Unbiased coverage, 2013. URL http://www.pacb.com/uncategorized/new-data-release-arabidopsis-assembly/.

[10] Pacific Biosciences. Falcon assembly pipeline, 2017. URL https://github.com/PacificBiosciences/FALCON.

[11] Pacific Biosciences. Blasr output format, 2017. URL https://github.com/PacificBiosciences/blasr/wiki/Blasr-Output-Format.

[12] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[13] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[14] Stuart M Brown. Global vs local alginment, 0000. URL http://slideplayer.com/slide/4463200/.

[15] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[16] Mark J Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC Bioinformatics*, 13(1):238, 2012.

[17] Chen-Shan Chin, Paul Peluso, Fritz J Sedlazeck, Maria Nattestad, Gregory T Concepcion, Alicia Clum, Christopher Dunn, Ronan O'Malley, Rosa Figueroa-Balderas, Abraham Morales-Cruz, et al. Phased diploid genome assembly with single molecule real-time sequencing. *bioRxiv*, page 056887, 2016.

[18] Arthur L Delcher, Simon Kasif, Robert D Fleischmann, Jeremy Peterson, Owen White, and Steven L Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11): 2369–2376, 1999.

[19] Arthur L Delcher, Adam Phillippy, Jane Carlton, and Steven L Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic acids research*, 30(11):2478–2483, 2002.

[20] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, et al. Real-time dna sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[21] Francisco Fernandes and Ana T Freitas. slaMEM: efficient retrieval of maximal exact matches using a sampled LCP array. *Bioinformatics*, page btt706, 2013.

[22] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[23] Walter Gilbert and Allan Maxam. The nucleotide sequence of the lac operator. *Proceedings of the National Academy of Sciences*, 70(12):3581–3584, 1973.

[24] Genome Assembly Group. Quast flow, 2014. URL `http://compgenomics2015.biology.gatech.edu/index.php/Genome_Assembly_Group`.

[25] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[26] Paul Jaccard. Distribution de la flore alpine dans le bassin des dranses et dans quelques régions voisines. *Bull Soc Vaudoise Sci Nat*, 37:241–272, 1901.

[27] W James Kent. Blatthe blast-like alignment tool. *Genome research*, 12(4):656–664, 2002.

[28] Zia Khan, Joshua S Bloom, Leonid Kruglyak, and Mona Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.

[29] Nilesh Khiste and Lucian Ilie. Laser: Large genome assembly evaluator. *BMC research notes*, 8(1):709, 2015.

[30] Nilesh Khiste and Lucian Ilie. E-MEM: efficient computation of maximal exact matches for very large genomes. *Bioinformatics*, 31(4):509–514, 2015.

[31] Nilesh Khiste and Lucian Ilie. Hisea: Hierarchical seed aligner for pacbio data. *BMC bioinformatics*, 18(1):564, 2017.

[32] Sergey Koren, Michael C Schatz, Brian P Walenz, Jeffrey Martin, Jason T Howard, Ganeshkumar Ganapathy, Zhong Wang, David A Rasko, W Richard McCombie, Erich D Jarvis, et al. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nature biotechnology*, 30(7):693–700, 2012.

[33] Sergey Koren, Brian P Walenz, Konstantin Berlin, Jason R Miller, and Adam M Phillippy. Canu: scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *bioRxiv*, page 071282, 2016.

[34] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome biology*, 5(2):R12, 2004.

[35] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, page btw152, 2016.

[36] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[37] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441, 1985.

[38] Bin Ma, John Tromp, and Ming Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[39] Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[40] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[41] Michael L Metzker. Sequencing technologiesthe next generation. *Nature reviews genetics*, 11(1):31–46, 2010.

[42] Giles Miclotte, Mahdi Heydari, Piet Demeester, Pieter Audenaert, and Jan Fostier. Jabba: Hybrid error correction for long sequencing reads using maximal exact matches. In *International Workshop on Algorithms in Bioinformatics*, pages 175–188. Springer, 2015.

[43] Eugene W Myers. An o (nd) difference algorithm and its variations. *Algorithmica*, 1(1): 251–266, 1986.

[44] Gene Myers. Efficient local alignment discovery amongst noisy long reads. In *International Workshop on Algorithms in Bioinformatics*, pages 52–67. Springer, 2014.

[45] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[46] Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *String Processing and Information Retrieval*, pages 347–358. Springer, 2010.

[47] Mihai Pop, Daniel S Kosack, and Steven L Salzberg. Hierarchical scaffolding with bambus. *Genome research*, 14(1):149–159, 2004.

[48] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20 (18):3363–3369, 2004.

[49] Michael G Ross, Carsten Russ, Maura Costello, Andrew Hollinger, Niall J Lennon, Ryan Hegarty, Chad Nusbaum, and David B Jaffe. Characterizing and measuring bias in sequence data. *Genome Biology*, 14(5):1, 2013.

[50] Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

[51] Frederick Sanger, Steven Nicklen, and Alan R Coulson. Dna sequencing with chain-terminating inhibitors. *Proceedings of the national academy of sciences*, 74(12):5463–5467, 1977.

[52] Grégory F Schneider and Cees Dekker. DNA sequencing with nanopores. *Nature biotechnology*, 30(4):326–328, 2012.

[53] Let's Talk Science's Education Services. Sanger sequencing, 2012. URL `https://explorecuriocity.org/Explore/ArticleId/2027/sanger-sequencing-2027.aspx`.

[54] Lloyd M Smith, Jane Z Sanders, Robert J Kaiser, Peter Hughes, Chris Dodd, Charles R Connell, Cheryl Heiner, Stephen BH Kent, and Leroy E Hood. Fluorescence detection in automated dna sequence analysis. 1986.

[55] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[56] Daniel D Sommer, Arthur L Delcher, Steven L Salzberg, and Mihai Pop. Minimus: a fast, lightweight genome assembler. *BMC bioinformatics*, 8(1):64, 2007.

[57] Ivan Sović, Mile Šikić, Andreas Wilm, Shannon Nicole Fenlon, Swaine Chen, and Niranjan Nagarajan. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nature communications*, 7, 2016.

[58] Martin Vingron and Michael S Waterman. Sequence alignment and penalty choice: Review of concepts, case studies and implications. *Journal of molecular biology*, 235(1): 1–12, 1994.

[59] Michaël Vyverman, Bernard De Baets, Veerle Fack, and Peter Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29 (6):802–804, 2013.

[60] Shih-Hao Wang and Yuan-Pei Lin. Wavelet tree quantization for copyright protection watermarking. *IEEE Transactions on Image Processing*, 13(2):154–165, 2004.

Appendix A

E-MEM Results For MEM Computation

A.1 Human vs Mouse

MEM computation results for Human vs Mouse.

A.1.1 Minimum MEM length 100

Table A.1: *Homo sapiens* vs *Mus musculus*; MEMs of minimum length 100.

Program		Time (s)		Memory (MB)	
		serial	12 cores	serial	12 cores
essaMEM	$K = 1$	–	–	–	–
	$K = 2$	4,076	3,107	19,065	19,697
	$K = 4$	8,291	3,174	11,264	11,896
	$K = 8$	9,243	2,069	7,394	8,282
	$K = 16$	4,437	1,385	5,468	6,394
	$K = 32$	6,245	3,520	4,508	5,396
	$K = 64$	9,044	5,119	4,029	4,917
slaMEM		62,099	–	3,480	–
sparseMEM	$K = 1$	–	–	–	–
	$K = 2$	3,845	2,947	20,182	20,750
	$K = 4$	19,797	6,833	12,426	13,250
	$K = 8$	58,325	10,217	8,548	9,327
	$K = 16$	50,703	9,169	6,609	7,497
	$K = 32$	46,492	8,853	5,640	6,528
	$K = 64$	41,396	9,398	5,155	6,043
Vmatch		7,370	–	39,377	–
E-MEM	$D = 1$	1,792	324	3,979	4,705
	$D = 2$	2,241	392	2,138	2,864
	$D = 3$	2,864	505	1,513	2,239
	$D = 4$	3,266	596	1,211	1,937
	$D = 5$	3,900	699	1,009	1,735
	$D = 6$	4,327	799	884	1,610
	$D = 7$	4,841	903	786	1,512
	$D = 8$	5,340	1,048	722	1,448
	$D = 9$	5,855	1,097	669	1,395
	$D = 10$	6,296	1,209	623	1,349

A.1.2 Minimum MEM length 300

Table A.2: *Homo sapiens* vs *Mus musculus*; MEMs of minimum length 300.

Program		TIME (s)		SPACE (MB)	
		serial	12 cores	serial	12 cores
essaMEM	$K = 1$	–	–	–	–
	$K = 2$	4,004	3,095	19,065	19,697
	$K = 4$	3,217	3,497	11,264	11,998
	$K = 8$	3,325	1,150	7,394	8,282
	$K = 16$	3,884	1,408	5,468	6,356
	$K = 32$	6,399	3,618	4,508	5,396
	$K = 64$	5,739	4,671	4,029	7,314
slaMEM		68,727	–	3,480	–
sparseMEM	$K = 1$	–	–	–	–
	$K = 2$	18,245	2,922	20,874	20,814
	$K = 4$	40,490	6,768	12,426	13,314
	$K = 8$	89,314	10,200	8,548	9,436
	$K = 16$	82,083	9,218	6,737	7,497
	$K = 32$	70,068	9,053	6,370	6,528
	$K = 64$	68,637	9,467	6,370	6,043
Vmatch		10,217	–	40,675	–
E-MEM	$D = 1$	929	199	2,293	3,019
	$D = 2$	1,466	313	1,262	1,988
	$D = 3$	1,977	399	924	1,650
	$D = 4$	2,471	495	754	1,480
	$D = 5$	2,899	574	652	1,378
	$D = 6$	3,443	664	581	1,307
	$D = 7$	3,957	745	530	1,256
	$D = 8$	4,554	867	492	1,219
	$D = 9$	4,834	897	464	1,190
	$D = 10$	5,149	961	442	1,168

A.2 Human vs Chimp

MEM computation results for Human vs Mouse.

A.2.1 Minimum MEM length 100

Table A.3: *Homo sapiens* vs *Pan troglodytes*; MEMs of minimum length 100.

Program		Time (s)		Memory (MB)	
		serial	12 cores	serial	12 cores
essaMEM	$K = 1$	–	–	–	–
	$K = 2$	3,452	2,642	19,057	19,633
	$K = 4$	25,328	7,887	11,384	12,088
	$K = 8$	24,220	4,422	7,386	8,282
	$K = 16$	10,404	1,850	5,588	6,356
	$K = 32$	12,381	3,661	4,628	5,396
	$K = 64$	16,048	5,275	4,149	4,917
sparseMEM	$K = 1$	–	–	–	–
	$K = 2$	10,169	2,511	20,174	20,814
	$K = 4$	20,346	4,898	12,606	13,250
	$K = 8$	57,390	9,049	8,540	9,436
	$K = 16$	56,946	9,413	6,601	7,535
	$K = 32$	58,210	9,267	5,632	6,528
	$K = 64$	49,807	10,083	5,147	6,043
Vmatch		7,696	–	39,725	–
E-MEM	$D = 1$	7,611	1,992	4,123	4,849
	$D = 2$	6,780	1,933	2,210	2,937
	$D = 3$	6,838	2,022	1,562	2,288
	$D = 4$	7,056	2,136	1,247	1,973
	$D = 5$	7,772	2,250	1,039	1,765
	$D = 6$	8,077	2,408	908	1,634
	$D = 7$	8,868	2,487	807	1,533
	$D = 8$	9,018	2,630	741	1,467
	$D = 9$	9,436	2,707	686	1,412
	$D = 10$	10,014	2,795	638	1,364

A.2.2 Minimum MEM length 300

Table A.4: *Homo sapiens* vs *Pan troglodytes*; MEMs of minimum length 300.

Program		TIME (s)		SPACE (MB)	
		serial	12 cores	serial	12 cores
essaMEM	$K = 1$	–	–	–	–
	$K = 2$	8,393	2,527	19,193	19,697
	$K = 4$	15,578	2,982	11,502	11,960
	$K = 8$	13,383	1,879	7,520	8,282
	$K = 16$	20,600	2,118	7,314	6,356
	$K = 32$	19,317	4,207	7,314	5,396
	$K = 64$	11,351	4,678	7,314	4,917
sparseMEM	$K = 1$	–	–	–	–
	$K = 2$	11,675	2,361	20,874	20,942
	$K = 4$	33,141	5,315	12,554	13,122
	$K = 8$	71,839	8,967	8,548	9,372
	$K = 16$	70,550	9,410	6,609	7,497
	$K = 32$	69,767	9,381	5,766	6,528
	$K = 64$	57,938	10,337	5,283	6,043
slaMEM		75,705	–	51,628	–
Vmatch		9,431	–	40,675	–
E-MEM	$D = 1$	1,416	289	2,438	3,164
	$D = 2$	1,753	403	1,335	2,061
	$D = 3$	2,629	525	973	1,699
	$D = 4$	2,993	647	790	1,516
	$D = 5$	3,103	742	682	1,408
	$D = 6$	3,603	871	605	1,331
	$D = 7$	4,294	970	551	1,277
	$D = 8$	5,019	1,083	511	1,237
	$D = 9$	5,116	1,174	481	1,207
	$D = 10$	7,384	1,256	457	1,183

A.3 Triticum aestivum vs Triticum durum

MEM computation results for Triticum aestivum vs Triticum durum.

A.3.1 Minimum MEM length 100

Table A.5: *Triticum aestivum* vs *Triticum durum*; MEMs of minimum length 100.

Program		Time (s)		Memory (MB)	
		serial	12 cores	serial	12 cores
E-MEM	$D = 1$	1,073	352	5,847	6,573
	$D = 2$	1,610	462	3,105	3,831
	$D = 3$	2,388	611	2,216	2,942
	$D = 4$	2,830	724	1,700	2,426
	$D = 5$	3,472	849	1,426	2,152
	$D = 6$	4,095	947	1,237	1,963
	$D = 7$	4,690	1,069	1,108	1,834
	$D = 8$	5,175	1,186	1,008	1,734
	$D = 9$	5,664	1,314	900	1,626
	$D = 10$	6,429	1,413	926	1,652
Vmatch		6,932	–	56,987	–

A.3.2 Minimum MEM length 300

Table A.6: *Triticum aestivum* vs *Triticum durum*; MEMs of minimum length 300.

Program		TIME (s)		SPACE (MB)	
		serial	12 cores	serial	12 cores
E-MEM	$D = 1$	870	328	3,367	4,093
	$D = 2$	1,902	391	1,822	2,548
	$D = 3$	2,321	530	1,344	2,071
	$D = 4$	2,832	649	1,045	1,771
	$D = 5$	4,381	755	922	1,618
	$D = 6$	4,830	845	826	1,522
	$D = 7$	4,937	975	731	1,457
	$D = 8$	5,666	1,047	678	1,404
	$D = 9$	6,044	1,181	641	1,367
	$D = 10$	7,046	1,273	657	1,383
Vmatch		29,084	–	56,987	–

Curriculum Vitae

Name:	Nilesh Vinod Khiste
Post-Secondary Education and Degrees:	University of Western Ontario London, Canada 2013 - 2018 Ph.D. Indian Institute of Technology Delhi, India 2009 - 2009 Course in Low Power Design Banaras Hindu University Varanasi, India 1998 - 2000 M.Sc. Banaras Hindu University Varanasi, India 1995 - 1998 B.Sc.
Honours and Awards:	Ontario Graduate Scholarship (OGS) 2017-2018 Ontario Graduate Scholarship (OGS) 2016-2017 Graduate Student Teaching Award 2016 University of Western Ontario Texas Instrument Low Power Team Award 2011 Cadence Design Systems
Related Work Experience:	Teaching Assistant The University of Western Ontario 2013 - 2017 Research and Development Engineer Cadence Design Systems 2005 - 2013 Senior Software Engineer Tata Consultancy Services 2000 - 2005

Publications:

Khiste, N. and Ilie, L., 2017. HISEA: HIerarchical SEed Aligner for PacBio data. BMC bioinformatics, 18(1), p.564.

Khiste, N. and Ilie, L., 2015. LASER: Large genome ASsembly EvaluatoR. BMC research notes, 8(1), p.709.

Khiste, N. and Ilie, L., 2015. E-MEM: efficient computation of maximal exact matches for very large genomes. Bioinformatics (Oxford, England), 31(4), p.509.